12

MECHANICAL INTELLIGENCE: RESEARCH AND APPLICATIONS

Final Technical Report

Covering the Period 12 April 1976 through 9 October 1977

December 1977

Edited by: Earl D. Sacerdoti

With Contributions by: Richard E. Fikes, Gary G. Hendrix, Paul Morris, Earl D. Sacerdoti, Daniel Sagalowicz, Jonathan Slocum

Prepared for:

Defense Advanced Research Projects Agency
Arlington, Virginia 22209

Contract DAAG29-76-C-0012
ARPA Order No. 2894
Program Element Code 61101E

SRI International
333 Ravenswood Avenue
Menlo Park, California 94025
(415) 326-6200
Cable: STANRES, Menlo Park
TWX: 910-373-1246

SRI International

MECHANICAL INTELLIGENCE: RESEARCH AND APPLICATIONS

December 1977

Final Technical Report
Covering the Period 11 April 1976 through 9 October 1977
SRI International Project 4763

Edited By
Earl D. Sacerdoti

With Contributions by
Richard E. Fikes, Gary G. Hendrix, Paul Morris,
Earl D. Sacerdoti, Daniel Sagalowicz, and Jonathan Slocum

CONTRACT DAAG29-76-C-0012
ARPA Order No. 2894
Program Element Code 61101E

Effective Date:	1 December 1975
Expiration Date:	9 October 1977
Amount of Contract:	$1,076,446

Prepared for:

Defense Advanced Research Projects Agency
Arlington, Virginia 22209

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
ARLINGTON, VIRGINIA 22209

Approved by:

Peter E. Hart, Director
Artificial Intelligence Center

Earle D. Jones, Executive Director
Information Science and Engineering Division

SRI International

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER (9)
~~4. TITLE (and Subtitle)~~ (6) MECHANICAL INTELLIGENCE: RESEARCH AND APPLICATIONS		~~5. TYPE OF REPORT & PERIOD~~ COVERED Final Technical Report. ~~Covering the period~~ 12 April 1976 through 9 October 1977
		6. ~~PERFORMING~~ ORG. REPORT NUMBER SRI Project 4763
7. AUTHOR(s) (10) Edited by Earl D. Sacerdoti; contributions by Richard E. Fikes, Gary G. Hendrix, Paul Morris Daniel Sagalowicz, Jonathan Slocum		~~8. CONTRACT OR GRANT NUMBER(s)~~ (15) DAAG-29-76-C-0012, new ✓ ARPA Order ~~No.~~ 2694
9. PERFORMING ORGANIZATION NAME AND ADDRESS SRI International 333 Ravenswood Avenue Menlo Park, California 94025		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS Program Element Code 61101E
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency Arlington, Virginia 22207		~~12. REPORT DATE~~ (11) November 1977 — 13. NO. OF PAGES 165
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office) (12) 177p		~~15. SECURITY CLASS.~~ (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence, Natural Language Processing, Data Base Access, Interactive Query Languages, Data Base Query Languages, Decision Aids, Command and Control

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report summarizes the results of a research project whose goal is to develop computer systems that can provide easy access for nontechnicians to large, distributed data bases of information. Our goal has been to develop mechanisms for automating many of the detailed tasks that today are normally performed by a decision maker's technical staff. These include accepting a question, in natural (not necessarily grammatical) English, in the decision maker's own terms; planning a sequence of queries to various files to gather the requested information; developing the plan into a computer program or programs in the language of the data over

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410281 JB

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

base management system on which the data resides; transmitting the retrieval programs, and monitoring their execution; and composing the information retrieved into a suitable output format.

Our work is in support of the Advanced Command Control Architectural Testbed (ACCAT) program under the sponsorship of the Defense Advanced Research Projects Agency (ARPA). The ACCAT program is intended to provide a facility for transferring emerging information processing technology to Navy command and control applications. While the direct application of our work has been to develop prototype systems to aid in naval command and control, the software tools we have created and the concepts underlying them offer potential aid to decision makers in the other services, government, and industry as well.

We have focused our efforts along two mutually supporting lines of research. First, we have created a performance system, called LADDER (for Language Access to Distributed Data with Error Recovery), that carries out all of the functions listed above in at least rudimentary form. Our second line of research focuses on longer-term efforts to develop the techniques required to satisfy more fully the needs of decision makers.

This report is a compendium of revised versions of five technical papers, published within the last six months, describing our efforts. Section II presents an overview of our research effort. The following three sections describe the components of our LADDER performance system in detail. Section III discusses the development of the natural language front end for our query system. Section IV presents IDA, the component of LADDER that plans the access to the data. Section V describes FAM, the component responsible for establishing and maintaining access to the distributed data base. Section VI discusses the major results of our longer-term research effort: our knowledge representation scheme and a system for reasoning about facts stored according to that scheme. Finally, Section VII lists the publications and presentations by the project staff during the current project.

CONTENTS

ILLUSTRATIONS

Preceding Page BLANK - FILMED

I INTRODUCTION

A. DECISION AIDS FOR COMMAND AND CONTROL

Since early 1976 we have been been working on a family of computer systems intended to provide easy access for non-technicians to large, distributed data bases of information. Our goal has been to develop systems that take on the following kinds of tasks, which today are normally performed by a decision maker's technical staff:

Accept a question, in natural (not necessarily grammatical) English, in the decision maker's own terms.

Determine what information is being requested from the distributed data base.

Plan a sequence of queries to various files to gather the requested information.

Develop the plan into a computer program or programs in the language of the data base management system on which the data resides.

Initiate and maintain access to the computer or computers housing the data.

Transmit the retrieval programs, and monitor their execution.

Compose the information retrieved into a suitable output format.

Our work is being carried out in support of the Advanced Command Control Architectural Testbed (ACCAT) program under the sponsorship of the Defense Advanced Research Projects Agency (DARPA). The ACCAT program is intended to provide a facility for transferring emerging information processing technology to Navy command and control applications. While the direct application of our work has been to develop prototype systems to aid in naval command and control, the software tools we have created and the concepts underlying them offer potential aid to decision makers in the other services, government, and industry as well.

We have focused our efforts along two mutually supporting lines of research. First, we have created a performance system that carries out all of the functions listed above in at least rudimentary form. This system, called LADDER (for Language Access to Distributed Data with Error Recovery) has been installed in the ACCAT facility since January 1977 (within a week of the installation of the ACCAT facility itself) and is undergoing continuing extension and improvement. A version of the system is available for public access over the ARPANET at SRI's interactive computer facility.* Our strategy for the development of LADDER is to adapt and extend existing artificial intelligence techniques, keeping in mind the requirement that the system perform in real time. The system's capabilities are limited, not so much by what we know how to do, but by what we know how to do fast.

Our second line of research focuses on longer-term efforts to develop the techniques required to satisfy more fully the needs of decision makers. We feel, with others in the data base field,** that a data base query system should have access to a model of the real-world processes that affect the data in the data base. Such a model will have to be created out of a sufficiently rich set of primitives, and will have to be supported by a deductive mechanism that is both complete and efficient. To this end, we have extended a formalism, called "partitioned semantic networks," for representing diverse kinds of knowledge within a computer memory. In addition, we have created algorithms for reasoning about information stored in this format, and for incorporating information stored in other formats (for example, external data bases) in the reasoning process as well. During the forthcoming year, we will apply these techniques to the task of data base querying.

----

* Those interested in trying out LADDER should contact Earl Sacerdoti (SACERDOTI@SRI-AI) for access to it.

** See, for example, the "conceptual schema," as defined in the report by a subcommittee set up by the American National Standards Institute [1].

This report is a compendium of revised versions of five technical papers published within the last six months describing our efforts. Section II presents an overview of our research effort. The following three sections describe the components of our LADDER performance system in detail. Section III discusses the development of the natural language front end for our query system. Section IV presents IDA, the component of LADDER that plans the access to the data. Section V describes FAM, the component responsible for establishing and maintaining access to the distributed data base. Section VI discusses the major results of our longer-term research effort: our knowledge representation scheme and a system for reasoning about facts stored according to that scheme. Finally, Section VII lists the publications and presentations made by the project staff during the current project.

A number of appendices provide more detailed information for the interested reader. Appendix 1 contains an annotated transcript of a session with LADDER. Appendix 2 is a formal description of how to use the IDA program in isolation. Appendix 3 is a similar description for FAM. Appendix 4 contains a glossary of names and acronyms referred to in the text.

B. ACKNOWLEDGEMENTS

The work described in this report represents the joint efforts of an integrated, energetic group at SRI International. Members of this group have included, over the lifetime of the project, Richard E. Fikes (now at Xerox PARC), Koichi Furukawa (now at the Electrotechnical Laboratory in Tokyo), Gary G. Hendrix, Staffan Lof, Paul Morris (now at UC Irvine), Nils J. Nilsson, Gordon S. Novak, Jr., William H. Paxton, Jane J. Robinson, Daniel Sagalowicz, Jonathan Slocum, and B. Michael Wilber. Koichi Furukawa and Staffan Lof participated in the research program as International Fellows at SRI, under sponsorship of the Electrotechnical Laboratory in Tokyo, Japan, and the National Defense Research Institute in Sundbyberg, Sweden, respectively. The current version of the LADDER system uses LIFER, SRI's proprietary package of language definition and parsing routines.

Preceding Page Blank - Not Filmed

II OVERVIEW OF THE RESEARCH EFFORT

by Earl D. Sacerdoti

A. INTRODUCTION

Man's use of tools shapes his environment. Man's use of tools also shapes his behavior. As technology evolves more complex tools, the impositions these tools make on their users become more stringent. Although it is difficult to reproduce strings of ten digits, we learn to do it well, because the interface to the telephone system demands it. Although it is difficult to type very fast (the standard keyboard was originally designed to allow enough time between keystrokes to keep early typewriters from jamming), we train ourselves to use a suboptimal --indeed, subaverage-- arrangement of keys, because the interface to keyboard systems demands it.

As the amount of information moving across the man-machine interface increases, the impositions of machines on our behavior also increase. Since computers exchange large amounts of information with their human users, they place great impositions on us. A goal of research in artificial intelligence is to reduce the extent of these impositions, thus making the benefits of computer use more widely available.

One example of the imposition set by the computer arises in the area of management information systems. Imagine that a user in a decision-making role knows that his data base contains some information that pertains to a decision he must make. The user wishes to extract that information from the data base and restructure, summarize, or analyze it in some way. Ideally, the user would be able to interact with the computer in his own terminology and issue a request for the information he desires. But today's computer systems typically require a very stilted, formal mode of interaction. Even then, the user will be able to obtain only certain preprogrammed reports, and this is hardly what is needed for the typical decision maker in his role of managing by exception.

If the decision maker wants a new perspective on the information in the data base, he must call in a programmer who works with the data base regularly. The programmer carries in his head four kinds of knowledge that must be used in order to gather the desired information. First, he knows how to translate the request for information from the decision maker's terms into the terms of the data that is actually stored in the data base. Second, he is able to convert the request for data from the overall data base into a series of requests for particular items of data from particular files. Third, he knows how to translate the particular requests into programs or calls on the data base management system's primitives in order to actually initiate the appropriate computation. Fourth, he knows how to monitor the execution of his request to ensure that the expected data is being obtained.

The need to automate the activities carried out by our hypothetical data base expert has been cited with increasing frequency, not only by those with a background in artificial intelligence, but also by computer scientists primarily concerned with data bases [8] [12] [37].

For the past year, a group at SRI has been working on just such an approach to data base access. The following subsection presents an overview of a running system that performs at least some of the expert's functions reliably and efficiently. Our current progress on representing and using each of the four kinds of knowledge described above will be detailed in the remainder of this section.

B. OVERVIEW OF THE LADDER SYSTEM

Our running demonstration system, called LADDER (for Language Access to Distributed Data with Error Recovery), represents an application of state-of-the-art techniques from the field of artificial intelligence in a real-time performance system. Because it consists of a number of rather independent, modular components, new capabilities can be incorporated easily as we learn how to make them run efficiently.

LADDER has been developed as a management aid to Navy decision makers, so the examples presented throughout this section and the following three are drawn from the domain of Navy command and control. Applications of this work to other decision-making and data access problems should be obvious.

The LADDER system consists of three major functional components, as displayed in Figure 1, that provide levels of buffering of the user from a data base management system (DBMS). LADDER employs the DBMS to retrieve specific field values from specific files just as a programmer might, so that the user of LADDER need not be aware of the names of specific fields, how they are formatted, how they are structured into files, or even where the files are physically located. Thus, the user can think he is retrieving information from a "general information base" rather than retrieving specific items of data from a highly formatted, traditional data base.

LADDER's first component accepts queries in a restricted subset of natural language. This component, called INLAND (for Informal Natural Language Access to Navy Data) produces a query or queries to the data base as a whole. The queries to the data base refer to specific fields, but make no mention of how the information in the data base is broken down into files.

For example, suppose a user types in "What is the length of the Kennedy?" (or "Give me the Kennedy's length," or even "Type length Kennedy"). INLAND would translate this into the query:

```
((? LGH) (NAM EQ 'JOHN#F.KENNEDY ')),
```

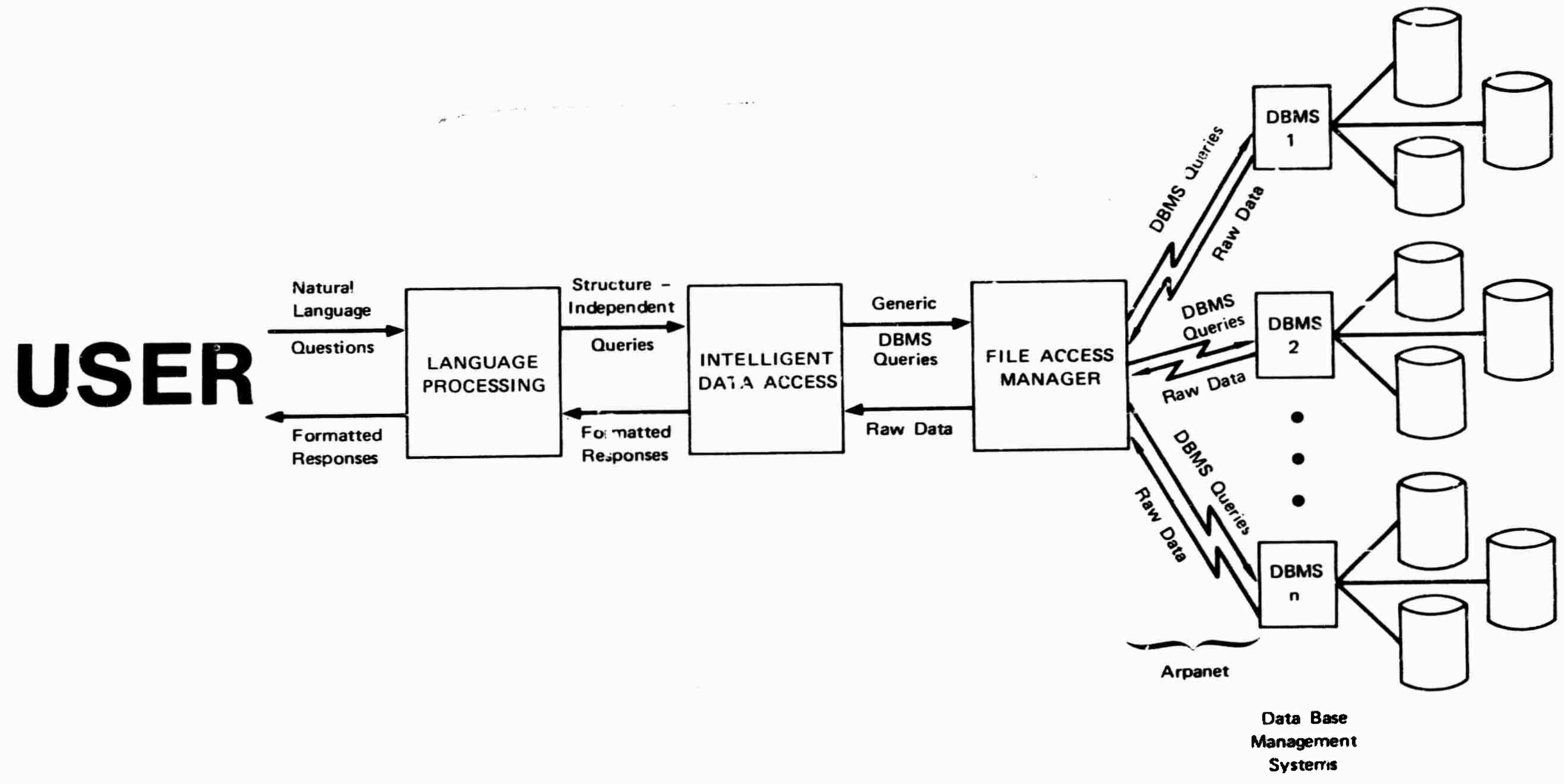


FIGURE 1 OVERVIEW OF THE LADDER SYSTEM

where LGH is the name of the length field, NAM the name of the ship name field, and 'JOHN#F.KENNEDY' the value of the NAM field for the record concerned with the Kennedy. This query is then passed along to the second component of the system.

The queries from INLAND to the data base are specified without any presumption about the way the data is broken up into files. The second functional component, called IDA (for Intelligent Data Access) breaks down the query against the entire data base into a sequence of queries against various files. IDA employs a model of the structure of the data base to perform this operation, preserving the linkages among the records retrieved so that an appropriate answer to the overall query may be returned to the user.

For example, suppose the data base consists of a single file whose records contain the fields

```
(NAM CLASS LGH).
```

Then, to answer the data base query issued above, IDA can simply create one file retrieval query that says, in essence, "For the ship record with NAM equal 'JOHN#F.KENNEDY', return the value of the LGH field." Suppose, however, that the data base is structured in two files, as follows:

```
SHIP: (NAM CLASS ...)
CLASS: (CLASSNAME LGH ...)
```

In this case the single query about the Kennedy's length must be broken into two file queries. These would say, first, "Obtain the value of the CLASS field for the SHIP record with NAM equal 'JOHN#F.KENNEDY'." Then, "Find the corresponding CLASS record, and return the value of the LGH field from that record." Finally, IDA would compose an answer that is relevant to the user's query (i.e., it will return NAM and LGH data, suppressing the CLASS-to-CLASSNAME link).

In addition to planning the correct sequence of file queries, IDA must actually compose those queries in the language of the DBMS. Our current system accesses, on a number of different machines, a DBMS called the Datacomputer [16] [13], whose input language is called

Datalanguage. IDA creates the relevant Datalanguage by inserting field and file names into pre-stored templates. However, since the data base in question is distributed over several different machines, the Datalanguage that IDA produces does not refer to specific files in specific directories on specific machines. It refers instead to generic files, files containing a specific kind of record. For example, the queries discussed above might refer to the SHIP file rather than file SHIP.ACTIVE in directory NAVY on machine DBMS-3. It is the function of the third major component of LADDER to find the location of the generic files and manage the access to them.*

To carry out this function, the third component, called FAM (for File Access Manager) relies on a locally stored model showing where files are located throughout the distributed data base. When it receives a query expressed in generic Datalanguage, it searches its model for the primary location of the file (or files) to which that query refers. It then establishes connections over the ARPANET to the appropriate computers, logs in, opens the files, and transmits the Datalanguage query, amended to refer to the specific files that are being accessed. If, at anv time, the remote computer crashes, the file becomes inaccessible, or the network connection fails, FAM can recover and, if a backup file is mentioned in FAM's model of file locations, establish a connection to a backup site and retransmit the query.

The existing system, written in INTERLISP [42], can process a fairly wide range of queries against a data base consisting of some 14 files containing about 100 fields. Processing a typical question takes a very few seconds of cpu time on a DEC KA-10 computer. An annotated transcript of a session with the system is provided in Appendix 1.

Thus LADDER provides at least some of the functions of the hypothetical data base expert in each area of expertise mentioned in the previous subsection. The following subsections will provide more

----

* In the introduction we described four activities that our system would carry out, and here we are describing only three functional components. This is because the third activity, translating particular queries into the primitives of particular DBMSs, is shared between IDA and FAM.

detailed views of the demonstration programs and ongoing research efforts in each of these areas.

C. NATURAL LANGUAGE INTERFACE

The task of providing access to the data base in the decision maker's terms is served by a functional component that accepts typed English text as input and produces formal queries to the IDA component as output. In order to provide truly natural access, this component must allow each user to expand the language definition with his own idiosyncratic language use.

We are developing a family of language interface components with increasing generality and true "understanding" of the input. In this subsection we describe our initial performance system.

Our initial system is built around a package of programs for language definition and parsing called Language Interface Facility with Ellipsis and Recursion (LIFER) [23]. LIFER consists of a parser and a set of interactive functions for specifying a language fragment oriented toward access of an existing computer system. The language is defined by what may be viewed as a set of productions of the form

<u>metasymbol</u> => <u>pattern</u>, <u>expression</u>,

where <u>metasymbol</u> is a metasymbol in the language, <u>pattern</u> is a list of metasymbols and symbols in the language, and <u>expression</u> is a LISP expression whose value, when computed, is assigned as the value of the metasymbol.

The set of productions is used by LIFER to build internal structures, called <u>transition trees</u>, that represent the language defined.* The transition trees are then used to parse user inputs in a top-down, left-to-right order. The response of the system to a user's input is simply the evaluation of the response expression associated with the top-level pattern that matches the input, together with all the subsidiary response expressions associated with metasymbols contained in

----

* Transition trees are a simplification of Woods' augmented transition networks [49].

the expansion of the top-level pattern or any expansion of a higher-level metasymbol.

The most important feature of LIFER from the point of view of developing a rich and usable language definition is the ease with which the grammar can be updated and the consequent changes tested. The ease of altering the grammar is such that LIFER provides a facility for casual users to add paraphrases to the language definition, in English. For example, the user might type

DEFINE (? LENGTH KENNEDY) LIKE (WHAT IS THE LENGTH OF THE KENNEDY) .

Subsequently, the system will accept

? COMMANDER KITTY HAWK

and

? SPEED AND CURRENT POSITION SUBS WITHIN 400 MILES OF GIBRALTAR

and interpret them correctly. Questions 10 through 12 in Appendix 1 provide a further example.

The LIFER parser has a very powerful mechanism for processing elliptical inputs, as exemplified by questions 2 and 4 in Appendix 1. Simple kinds of anaphoric reference, such as that shown in question 9, are handled within the language definition.

The nature of the LIFER parser imposes a discipline on the developer of the language definition. For parsing to operate efficiently, the grammar must severely restrict the number of acceptable words at each point in a sentence, and the tests applied to words in the left-to-right scan must be as cheap as possible. These goals are best satisfied with a language definition that directly encodes into the syntax most of the restrictions imposed by the semantics of the domain. Rather than contain metasymbols like "noun phrase," the INLAND grammar is composed of entities like "ship specification," "carry-verb phrase," and "pair of positions." Question 6 in Appendix 1 gives an example of a small fragment of the INLAND grammar. This approach of producing a semantically-oriented syntax is similar to that used by Brown and Burton [6] [7] and Waltz [45] [46].

Using LIFER's interactive language definition facilities, we have developed a language definition that we believe is one of the most extensive that has been incorporated into a computer system. It accepts a wide range of queries about the information in the data base as well as queries about the definitions of data base fields and about the grammar itself.

D. INTELLIGENT DATA ACCESS

A casual user would like to be able to access a data base as if it were an unstructured mass of information. Unfortunately, a data base is in reality a collection of files, often with very complex linkages among them. Even worse, a distributed data base may consist of different files on different machines, possibly handled by different DBMSs. An operation amounting to automatic problem solving is required to decide how to link up the files in the data base to extract and aggregate the information requested in a given query. An example of this situation in our demonstration system is a question such as "What is the fastest ship within 500 miles of Naples?" This single question from the user's point of view requires four queries of three files to develop an answer.

Our initial efforts in this area have concentrated on access planning for collections of data bases supporting a relational model of the data [11]. The knowledge necessary to decide how to link among relations is contained in what we call a <u>structural schema</u>. The structural schema contains information for each relation describing how it can be linked to other relations. In addition, it contains information about each field's counterparts in other relations and certain special-case information.

We have taken two approaches to the process of intelligent data access. The first, embodied in a program called IDA [39], uses a heuristic approach to the problem of linking among files. The structural schema is embodied in a frame-like representation [30] with individual frames defined for each field and each file. The program generates a single query at a time, examines the results, and then determines the next query to be asked. This approach can lead to suboptimal sequences of file accesses or can even fail to answer an answerable question, but it trades these shortcomings for rapid execution and straightforward extensibility.

Our second approach, embodied in a design for a program called DBAP (for Data Base Access Planner) [18], uses a formal, theorem-proving approach. The structural schema is represented as a set of axioms about the elements in the query language, the fields, and the files. These axioms are encoded as QLISP [47] procedures. The program builds a complete sequence of queries to the data base before beginning the actual interactions with it. Thus, it can plan an optimal sequence of file accesses, given a sufficiently detailed model of the data base. A partial implementation indicates that this approach is essentially an order of magnitude slower than IDA. For very large files this expenditure of planning time would undoubtedly be repaid by faster data base retrieval.

E. FILE ACCESS MANAGEMENT

The third major component of LADDER, called FAM (for File Access Manager) [32], locates particular files within the distributed data base, establishes connections to them, and transmits to and monitors the responses from the remote computers where the files are located. FAM can recover from a range of expected types of errors by establishing links to backup files and retransmitting the failed query.

FAM accepts as input Datalanguage commands that refer not to specific files on specific machines, but to generic files, as defined in subsection B above. Based on a locally stored model of the distributed file system, FAM selects the appropriate specific files for the generic files mentioned in the commands. If network links to the machines where the files reside do not yet exist, they are established. If the files in question are not yet open, they are opened. Finally, the query, modified to refer to specific file names, is transmitted to the remote machine.

If certain types of errors occur during the prosecution of the query, FAM will attempt to recover. FAM currently handles two types of error conditions. The first is a failure of the network connection, which is usually noticed by LADDER's host operating system (TENEX or TOPS-20) as a lack of interaction over the network for a given interval of time. In this case, FAM attempts to find alternative locations for the files referenced in the query, establishes links to them, and retransmits the query. The second type of error is an explicit complaint from the Datacomputer. In practice, this usually arises when FAM's model is inaccurate, and a file that was expected to be in a particular location in fact was not. In this case, FAM updates its model and attempts to recover as before.

FAM is implemented by making strong use of the features of INTERLISP that support multiple control and access environments [42] [3]. When FAM opens a connection to a particular machine, it builds a piece of pushdown stack that contains as locally bound variables the appropriate information about that connection, and whose control environment is poised to interact with the remote machine. An interaction with a particular remote machine can thus be invoked via a generator function.

F. SUMMARY AND DIRECTIONS FOR FURTHER WORK

As of October 1977, the LADDER system has been brought to a stage of development where it can be used with some success and enjoyment by casual users. It accepts a rather wide range of queries against a simple data base and is quite robust. The three major components of LADDER each address separate portions of the data access problem. Although they have been designed to work in combination, each component is a separate, self-contained module that independently addresses one aspect of data access. For example, the virtual view of the data that IDA supports for its caller would be of value even without a natural language front end. Likewise, the general technology developed for natural language translation may be separated from the data access problem and applied in other domains.

The level of performance we have achieved has been reached by making many simplifying assumptions. The language component does not understand the user's queries in any fundamental sense; rather, it reflexively invokes IDA with the appropriate arguments. The data access component assumes that all queries can be answered by joining records from various files. Both systems make strong assumptions that the user knows the kinds of information that are in the data base and is asking relevant questions. Now that an initial system has been developed and demonstrated, we can concentrate on efforts to improve its robustness, generality, and coverage of the language.

Until recently, there existed a clear trade-off between building two kinds of language systems. On the one hand, systems existed that ran reliably in real time but had very meager semantic underpinnings, whose extensibility was clearly limited, and which did not truly understand inputs to them, in the sense that they did not compose an internal representation of their meanings. On the other hand, there were systems that covered the language much more thoroughly, were better grounded linguistically, and developed a representation of what the inputs meant, but that could not run in real time. Even worse, there was no clear way to integrate the efforts being put into the two approaches: the underlying control structures and language definition systems were incompatible.

After evaluating the benefits of the LIFER approach and reexamining the requirements and behavior of the more semantically based systems, we have developed a "core language system" that is capable of supporting both approaches, and of supporting systems at intermediate positions on the tradeoff between real-time performance and linguistic grounding.

The core system accepts a wide range of styles of language definition, ranging from the semantically oriented syntax of the INLAND grammar to an amalgam of multiple knowledge sources similar to that used by the SRI speech understanding system [44]. It accepts language definitions at intermediate points within that range as well, and it should thus constitute a vehicle for bringing more linguistically and semantically oriented styles of language processing into actual use in a staged fashion. We are developing a research plan that should enable us to simultaneously explore the issues involved in true language understanding while augmenting the power, coverage, and linguistic relevance of the demonstration system.

Our plans for data access include extensions to the input language of IDA to permit quantified queries. This will enable the system to distinguish between such queries as "What is the last reported position of each sub?" and "What is the last reported position of any sub?"

We will attempt to demonstrate the generality of our approach to data base access planning by interfacing to a distributed data base stored partially on Digital Equipment Corporation's DBMS-20 [14], which supports a CODASYL-type [10] data model and partially on the Datacomputer, which supports a relational model.

In addition to these efforts, which we expect will improve our performance system, we are continuing to progress in our longer-range research. An integrated language understanding and access planning system built around the representation of knowledge in semantic network form is being designed. The longer-term efforts will benefit from the tool-building involved in the performance-oriented work. Development of the performance system is guided and prioritized by the results and problems encountered in our longer-term research. The early successes of this program have provided an initial demonstration of the benefits of simultaneously pursuing lower-risk research aimed at cost-effective performance and higher-risk research aimed at advancing the state of the art.

Preceding Page Blank - Filmed

III DEVELOPING A NATURAL LANGUAGE INTERFACE TO COMPLEX DATA

by Gary G. Hendrix, Earl D. Sacerdoti,
Daniel Sagalowicz, and Jonathan Slocum

A. THE NATURAL LANGUAGE COMPONENT

With the goal of supplying natural language interfaces to a variety of computer software, we developed in late 1975 a language processing package called LIFER (for Language Interface Facility with Ellipsis and Recursion) [23] that facilitates the construction and run-time operation of special-purpose, applications-oriented, natural language interfaces. INLAND (for Informal Natural Language Access to Navy Data), the linguistic component of our LADDER system for access to distributed data, has been constructed within the LIFER framework. Table 1 gives some indication of the diversity of language accepted by this system. Below we describe the nature of INLAND and illustrate how it was created using LIFER's interactive language definition facilities. The examples, of course, can show only limited aspects of INLAND. We believe the existing INLAND system to be one of the most robust computerized natural language systems ever developed, accepting a wide range of questions about information in the data base (as shown in Table 1) as well as metaquestions about definitions of data base fields and tne grammar itself.

Table 1. A Sample of Acceptable Inputs to LADDER

What kind of information do you know about

Is there a doctor on board the Biddle

Display all the American cruisers in the North Atlantic

What is the name and location of the carrier nearest to New York

What is the commanding officer's name

Who commands the Kennedy

What is the Kennedy's beam

When will the Los Angeles reach Norfolk

Tell me when Taru is scheduled to leave port

Where is she scheduled to go

When will Los Angeles arrive in its home port

When will the Sturgeon arrive on station

What aircraft units are embarked on the Constellation

To which task organization is Knox assigned

Where is the Sellers

Where is Luanda

What is the next port of call of the Santa Inez

When will Tarifa get underway

Which convoy escorts have inoperative sonar systems

When will they be repaired

Which US Navy DDGs have casreps involving radar systems

What Soviet ship has hull number 855

To what class does the Soviet ship Minsk belong

What class does the Whale belong to?

What is the normal steaming time for the Wainwright from Gibraltar to Norfolk

What American ships are carrying vanadium ore

How far is it to Norfolk

How far away is Norfolk

How many nautical miles is it to Norfolk

How many miles is it to Norfolk from here

How close is the Baton Rouge to Norfolk

How far is the Adams from the Aspro

What is the distance from Gibraltar to Norfolk

What is the nearest oiler

What is the nearest oiler to the Constellation

How far is it from Naples to 23-00N, 45-00W

What is the distance from the Kittyhawk to Naples

How long would it take the Independence to reach 35-00N, 20-00W

How long is the Philadelphia

How long would it take the Aspro to join Kennedy

What is the nearest ship to Naples with a doctor on board

What is the nearest USN ship to the Enterprise with an operational air search radar

What is known about that ship

How many merchant ships are within 400 miles of the Hepburn

What are their identities and last reported locations

What cargo does the Pecos have

Who is CTG 67.3

What are the length, width, and draft of the Kitty Hawk

To whom is the Harry E Yarnell attached

What type ships are in the Knox class

Where are the Charles F. Adams class ships

What are their current assignments

What subs in the South Atlantic are within 1000 miles of the Sunfish

What is the Kittyhawk doing

How many USN asw capable ships are in the Med

Where are they

What are their current assignments and fuel states

What ships are NOT at combat readiness rating C1

When will Reeves achieve readiness rating C1

Why is Hoel at readiness rating C2

When will the sonar be repaired on the Sterett

What ships are carrying cargo for the United States

Where are they going

What are they carrying

When will they arrive

Where is Gridley bound

Which cruisers have less than 50 per cent fuel on board

Where are all the merchant ships

When will the Kitty Hawk's radar be up?

What ships are in the Los Angeles class

What command does Adm. William have

Under whose opcon is the Dale

Show me where the Kennedy is!

What ship has hull number 148?

What is the next port of call for the South Carolina?

Are doctors embarked in the Kawishiwi

What kind of cargo does the Francis McGraw have?

What air group is embarked in the Constellation?

What do you know about the employment schedule of the Lang?

Which systems are down on the Kitty Hawk

What ships in the Med have doctors embarked?

How many ships carrying oil are within 340 miles of Mayport?

What sub contacts are within 300 miles of the Enterprise?

List the current position and heading of the US Navy ships in the Mediterranean every 4 hours

What is the status of the Enterprise's air search radar?

Where is convoy NL53 going

What convoy is the Transgermania in

How many embarked units are in Constellation

What ships are in British ports

What U S ships are within 500 miles of Wilmington?

What US ships faster than the Gridley are in Norfolk

What is the fastest ship in the Mediterranean Sea

How close is that ship to Naples?

What is its home port

Print the American cruisers' current positions and states of readiness!

How is the Los Angeles powered

What ship having a normal cruising speed greater than 30 knots is the largest

Display the last reported position of all ships that are in the North Atlantic

When did the Endeavour depart the port of New York

What nationality is the ship with international radio call sign UA1D

What ports are in the data base

What merchant ships are enroute to New York and within 500 miles of the Saratoga

To what country does the fastest sub belong?

1. Overview of LIFER

Although work in artificial intelligence and computational linguistics has not yet developed a general approach to the problems of understanding English and other natural languages, mechanisms do exist for dealing with major fragments of language pertinent to particular application areas. The idea behind LIFER is to adapt existing computational linguistic technology to practical applications while investigating and extending the human engineering aspects of the technology. The LIFER system supplies basic parsing procedures and an interactive methodology needed by a system developer to create convenient interfaces (such as INLAND) in reasonable amounts of time. Certain user-oriented features, such as spelling correction, processing of incomplete inputs, and the ability of the run-time user to extend the language accepted by the system through the use of paraphrase, are also included in the LIFER package.

LIFER is composed of two basic parts: a set of interactive language specification functions and a parser. The language specification functions are used to define an application language, a subset of a natural language (e.g., English) that is appropriate for interacting with existing software, such as a DBMS. Using this language specification, the LIFER parser interprets natural language inputs, translating them into appropriate interactions with the application software.

FIGURE 2 SIMPLIFIED INTERACTIONS WITH LADDER

```
1-What is the length of the Constellation
PARSED!
(LENGTH 1072 feet)

2-of the Nautilus
TRYING ELLIPSIS:  WHAT IS THE LENGTH OF THE NAUTILUS
(LENGTH 319 feet)

3-displacement
TRYING ELLIPSIS:  WHAT IS THE DISPLACEMENT OF THE NAUTILUS
(STANDARD-DISPLACEMENT 4040 tons)

4-length of the fastest American Nuclear sub
TRYING ELLIPSIS:  WHAT IS THE LENGTH OF THE FASTEST AMERICAN NUCLEAR SUB
(LENGTH 360 feet NAM LOS ANGELES  SPEED 30.0 knots)

5-Who commands the Constallation
        SPELLING-->CONSTELLATION
PARSED!
(COMMANDER  CAPT J.ELLISON)

6-Who commands JFK
TRYING ELLIPSIS:  ELLIPSIS HAS FAILED
THE PARSER DOES NOT EXPECT THE WORD "JFK" TO FOLLOW
"WHO COMMANDS"
OPTIONS FOR NEXT WORD OR META-SYMBOL ARE:
<SHIP-NAME>

7-Define JFK to be like Kennedy
PARSED!
 .  {JFK is now a synonym for KENNEDY, which is a ship name}
 .
 .

8-Who commands JFK   {that is, retry interaction 6}
PARSED!
(COMMANDER CAPT P.MOFFETT)

9-info JFK country
TRYING ELLIPSIS:  ELLIPSIS HAS FAILED
 .  {error message omitted}
 .
1
10-Define "Info JFK country" to be like "what is the country of JFK"
PARSED!
 .
 .
```

```
11-Info JFK country
PARSED!
(NATION USA)

12-Info fastest American nuclear submarine speed
PARSED!
(SPEED 30.0 knots   NAM LOS ANGELES)

13-Nautilus
TRYING ELLIPSIS:  INFO NAUTILUS SPEED
(SPEED 22 knots)
```

Figure 2. Simplified Interactions with LADDER

Figure 2 shows simplified example interactions with the LIFER parser using the INLAND language specification. A sequence of complete examples is presented in Appendix 1. The user of the system types in a question or command in ordinary English, followed by a carriage return. The LIFER parser then begins processing the input. When analysis is complete, the system types "PARSED!" and invokes data base functions (IDA) to respond.

An important feature of the parser is an ability to process elliptical (incomplete) inputs. Thus, if the system is asked, as in question 1 of Figure 2,

WHAT IS THE LENGTH OF THE CONSTELLATION

then the subsequent input

OF THE NAUTILUS

will be interpreted as WHAT IS THE LENGTH OF THE NAUTILUS.

If a user misspells a word, LIFER attempts to correct the error, using the INTERLISP spelling corrector [42]. If the parser cannot account for an input in terms of the application language definition, error messages, such as that produced after question 6, are printed that indicate how much of the input was understood and that suggest means of completing the input.

Provision is included in INLAND for interfacing with LIFER's own language specification functions, making it possible for users to give natural language commands for extending the language itself. In particular, computer naive users may extend the language accepted by the system by employing easy-to-understand notions such as synonyms and paraphrases. This is illustrated by interactions 7 and 10.

In using LIFER to define a language for INLAND, we have followed the approach taken by most real-time language processing systems in embedding considerable semantic information in the syntax of the language. Such a language specification is typically called a "semantic grammar." For example, words like NAUTILUS and DISPLACEMENT are not grouped together into a single <NOUN> category. Rather, NAUTILUS is treated as a <SHIP-NAME> and DISPLACEMENT as an <ATTRIBUTE>. Similarly, very specific sentence patterns such as

WHAT IS THE <ATTRIBUTE> OF <SHIP>

are typically used instead of more general patterns such as

<NOUN-PHRASE> <VERB-PHRASE>.

For each syntactic pattern, the language definer supplies an expression for computing the interpretation of instances of the pattern. INLAND's expressions for sentence-level patterns usually invoke the IDA component to retrieve information from the distributed data base.

This method of language specification is easy to understand and easy to use. But, when pursued systematically, it allows languages of rather broad coverage to be defined, as indicated in Table 1.

To provide a more detailed view of how LIFER has been employed to produce an efficient and effective language processing system, let us examine in detail a highly simplified fragment of the INLAND language specification.

2. INLAND's Function in Brief

The central notions of how INLAND is constructed may be seen by considering the problem of providing English access to two files of the form

```
SHIP:(NAM CLASS COMMANDER HOME-PORT HULL# LOC)
CLASS: (CLASSNAME TYPE NATION FUEL LENGTH BEAM DRAFT SPEED)
```

located on different computers. IDA and FAM together provide levels of insulation from the real situation, so that INLAND need consider only the problem of specifying what subset of the overall data base should be queried, and what field values within that subset should be returned. IDA will dynamically plan the appropriate joins on the files in the data base, and FAM will carry them out.*

3. A Miniature Language Specification

a. Productions

The grammar rules may be viewed as productions of the form

metasymbol => pattern ¦ expression,

where metasymbol is a metasymbol of the application language, pattern is a list of symbols and metasymbols in the language, and expression is a LISP expression whose value, when computed, is assigned as the value of the metasymbol.** The symbol <L.T.G> (LIFER Top Grammar) is the highest-level metasymbol of the grammar. The system's answer to complete inputs that match a pattern instantiating <L.T.G> will be the result of evaluating the associated LISP expression.

For example, the input

PRINT THE LENGTH OF THE KENNEDY

is an instantiation of the sentence-level production

```
 <L.T.G>  =>  <PRESENT> THE <ATTRIBUTE> OF <SHIP> ¦
              (IDA (APPEND <SHIP> <ATTRIBUTE>)).
```

The input matches the pattern

<PRESENT> THE <ATTRIBUTE> OF <SHIP>,

where <PRESENT> matches PRINT, <ATTRIBUTE> matches LENGTH, and <SHIP>

----

* In order to make the language processing issues clearer, we have suppressed detail in this section involving other aspects of the LADDER system. In particular, in the actual LADDER system, the intertwining of multiple files is much more complex than in the current example, and the output of INLAND (and hence the input to IDA) is slightly more complex than shown here.

** In addition to computing values for acceptable applications of the production, the expression may also be used to reject some applications on semantic grounds. Rejection is signaled if the expression returns *ERROR* as its value.

matches the phrase THE KENNEDY. If the semantic values for <SHIP> and <ATTRIBUTE>, computed by means to be described shortly, are ((NAM EQ JOHN#F.KENNEDY)) and ((? LENGTH)), respectively, then the answer to the question is computed from the expression portion of the production as follows:

```
              (IDA (APPEND <SHIP> <ATTRIBUTE>))
          =>  (IDA (APPEND '((NAM EQ JOHN#F.KENNEDY)) '(? LENGTH)))
          =>  (IDA '((NAM EQ JOHN#F.KENNEDY)(? LENGTH))).
```

(APPEND is a LISP function that appends any number of lists together to form a larger list.) At this point, the IDA component is called with the argument

```
                 ((NAM EQ JOHN#F.KENNEDY) (? LENGTH))
```

and the length of the Kennedy is retrieved:

```
            (IDA  '((NAM EQ JOHN#F.KENNEDY) (? LENGTH)))
         => (LENGTH 1072 feet)
```

In LIFER, productions like the one shown above are defined interactively by issuing commands such as

```
PD[<L.T.G>
   (<PRESENT> THE <ATTRIBUTE> OF <SHIP>)
   (IDA (APPEND <SHIP> <ATTRIBUTE>))],
```

where PD is the production definition function.

b. Lexical Entries

Metasymbols, such as <PRESENT> and <ATTRIBUTE>, are often associated with individual words or fixed phrases, which are maintained in LIFER's lexicons. The LIFER function MS (Make Set) is used to define a set of words and phrases that may match a particular metasymbol. For example, the call

```
MS[<ATTRIB>
   (CLASS COMMANDER FUEL TYPE NATION LENGTH
   BEAM DRAFT (LOCATION . LOC) (POSITION . LOC)
   (NAME . NAM) (COUNTRY . NATION)
   (NATIONALITY . NATION)((HOME PORT) . HOME-PORT)
   ((POWER TYPE) . FUEL)((HULL NUMBER) . HULL#))]
```

is used to define 16 words and fixed phrases that may match the symbol <ATTRIB> (which will be used subsequently in defining <ATTRIBUTE>).

After this call to MS, <ATTRIB> will match the words CLASS, COMMANDER, FUEL, TYPE, NATION, LENGTH, BEAM, and DRAFT. For these words, <ATTRIB> will take as its semantic value the word itself. <ATTRIB> will also match the word LOCATION, but for this match the value of <ATTRIB> will be LOC. Similarly, <ATTRIB> matches POSITION, NAME, COUNTRY, and NATIONALITY but takes the values LOC, NAM, NATION, and NATION, respectively. <ATTRIB> also matches the two-word phrase HOME PORT, taking HOME-PORT as its value. For the phrase POWER TYPE, the value is FUEL; for HULL NUMBER it is HULL#. (It is assumed that the codes HOME-PORT, HULL#, LOC, and NAM are peculiar to the data base and will not occur in natural language inputs.)

c. Subgrammars

Metasymbols may also be defined by production rules. For example, the call

```
PD[<ATTRIBUTE>
   (<ATTRIB>)
   (LIST (LIST '? <ATTRIB>))]
```

indicates that an <ATTRIBUTE> may be matched by an <ATTRIB>, viz.:

```
<ATTRIBUTE> => <ATTRIB>.
```

For this production, the associated expression is

```
(LIST (LIST '? <ATTRIB>)).
```

Since the word LENGTH matches <ATTRIB> and causes <ATTRIB> to take LENGTH as its value, the rule above indicates that LENGTH is an instantiation of <ATTRIBUTE>. That is

```
<ATTRIBUTE> => <ATTRIB> => LENGTH.
```

The value assigned to <ATTRIBUTE> when it matches LENGTH is computed by the production's expression as follows:

```
   (LIST (LIST '? <ATTRIB>))
=> (LIST (LIST '? 'LENGTH))
=> (LIST '(? LENGTH))
=> '((? LENGTH)).
```

This fragment of an IDA command requests the value of the LENGTH field. It was used above in answering the question "What is the length of the Kennedy?"

To recognize inputs such as

PRINT THE LENGTH BEAM AND DRAFT OF THE KENNEDY,

the concept of an <ATTRIBUTE> may be generalized* by adding two new productions as follows:

```
PD[<ATTRIBUTE>
   (<ATTRIB> AND <ATTRIBUTE>)
   (CONS (LIST '? <ATTRIB>) <ATTRIBUTE>)]

PD[<ATTRIBUTE>
   (<ATTRIB> <ATTRIBUTE>)
   (CONS (LIST '? <ATTRIB>) <ATTRIBUTE>)].
```

(CONS is a LISP function that adds an element, in this case the list whose first element is ? and whose second element is the value of <ATTRIB>, to the front of a list, in this case the value of <ATTRIBUTE>.) These productions allow the phrase LENGTH BEAM AND DRAFT to be accounted for in terms of the syntax tree of Figure 3.

d. Complete Analysis of a Simple Query

The examples above have indicated how the pattern

<PRESENT> THE <ATTRIBUTE> OF <SHIP>

may be defined as a top-level input and how the metasymbol <ATTRIBUTE> may be defined. To complete the analysis of the top-level pattern, consider now the following definitions for <PRESENT> and <SHIP>.

To define <PRESENT>, the function MS may be used:

```
MS[<PRESENT>
   (PRINT LIST SHOW GIVE ((GIVE ME) . PRINT)
   ((WHAT IS) . PRINT) ((WHAT ARE) . PRINT))].
```

This call allows <PRESENT> to match the words PRINT, LIST, SHOW, and GIVE, and the phrases GIVE ME, WHAT IS, and WHAT ARE. The values assigned to <PRESENT>, which might be used, for example, to direct

----

* The use of two symbols, <ATTRIB> and <ATTRIBUTE>, could be avoided by letting <ATTRIBUTE> directly match lexical items and by introducing such productions as

<ATTRIBUTE> => <ATTRIBUTE> AND <ATTRIBUTE>

Unfortunately, the collapse of the two symbols into one results in both ambiguity and left recursion. LIFER recognizes only one of the ambiguous interpretations. Left recursion can be tolerated by special mechanisms in LIFER's top-down, left-to-right parser, but only at a considerable increase in parsing time.

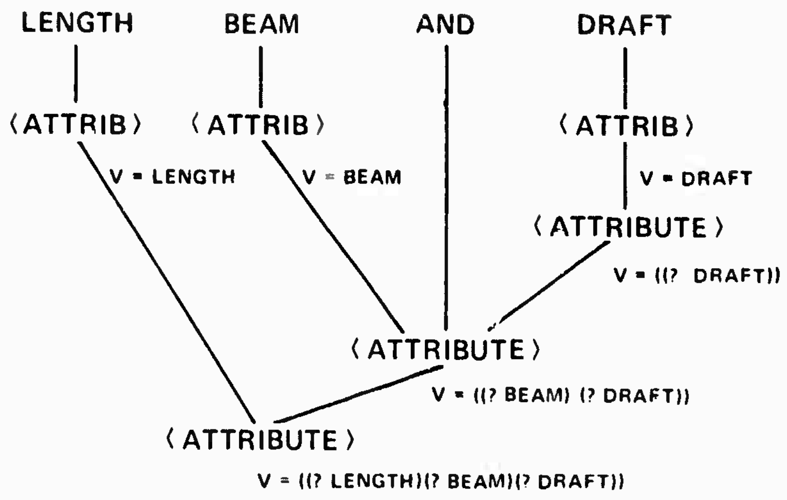


FIGURE 3 ⟨ATTRIBUTE⟩ SYNTAX TREE

output to the terminal or to a graphics subsystem, are not of interest here.

A <SHIP> may be designated in any one of a number of ways, the simplest being by name. The call

```
PD[<SHIP>
   (<SHIP-NAME>)
   (LIST (LIST 'NAM 'EQ <SHIP-NAME>))]
```

causes <SHIP> to match a <SHIP-NAME> and to take as its value an IDA command fragment restricting the value of the NAM field to be EQ (equal) to the particular name. <SHIP-NAME> may be defined by MS:

```
MS[<SHIP-NAME>
   (CONSTELLATION NAUTILUS
    (KENNEDY . JOHN#F.KENNEDY)
    ((JOHN F. KENNEDY) . JOHN#F.KENNEDY)  etc.)]
```

For an actual data base, this list is, of course, much more extensive. To allow the optional use of "the" before the name of a ship, a supplementary production for <SHIP> may be defined:

```
PD[<SHIP>
   (THE <SHIP>)
```

<SHIP>].

With these definitions, LIFER has been given all the information needed to process a small class of sentence-level inputs. For example, the complete analysis of the input

WHAT IS THE LENGTH OF THE KENNEDY

is shown in the syntax tree of Figure 4. Note how the query given to IDA was generated by combining fragments from <SHIP> and <ATTRIBUTE>.

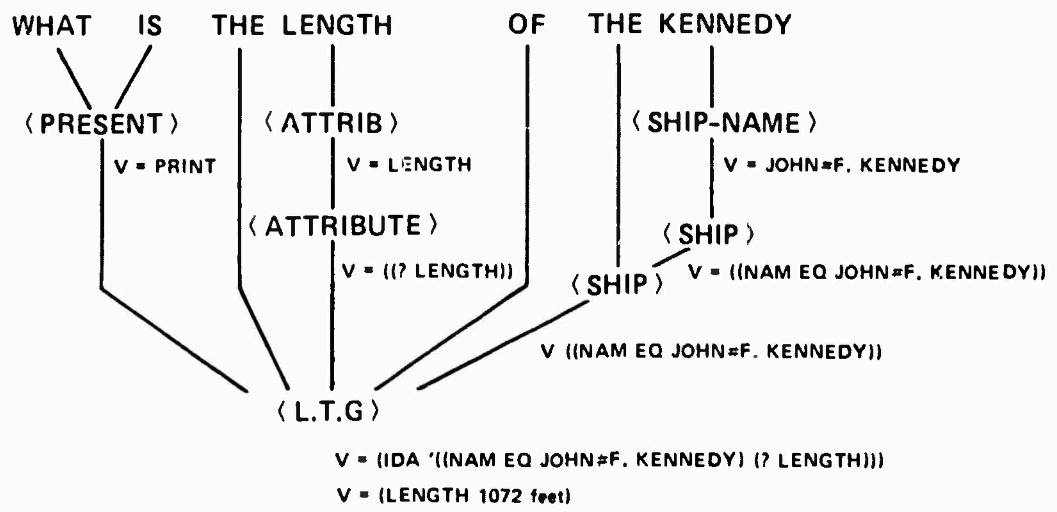


FIGURE 4 SYNTAX TREE FOR A COMPLETE QUESTION

From the definitions for complete inputs defined above, LIFER can infer how to process incomplete inputs in context. For example, having just parsed the input

WHAT IS THE LENGTH OF THE KENNEDY

the system may, without additional knowledge, also handle the following sequence of incomplete inputs:

BEAM
 (i.e., what is the beam of the Kennedy)
HOME PORT AND CLASS
 (i.e., what is the home port and class of the Kennedy,
NAUTILUS
 (i.e., what is the home port and class of the Nautilus).

The method by which these incomplete inputs are processed is discussed below.

Other inputs that the rules defined thus far will accept include:

```
GIVE ME THE POSITION OF THE NAUTILUS
PRINT THE HULL NUMBER AND POWER TYPE OF CONSTELLATION
SHOW THE COMMANDER COUNTRY AND TYPE OF THE JOHN F. KENNEDY .
```

4. Some Generalizations

The tiny fragment of language defined above already allows English access to most of the fields in the example data base, given the name of a ship. This fragment may be expanded easily along many dimensions.

a. Generalizing <SHIP>

Generalizing <SHIP> provides one of the most fruitful expansions. Naval ships are divided into major sets called classes. For example, the Constellation is in the Kitty Hawk class. Sometimes users will wish to ask questions about all ships of a particular class, e.g., HOW LONG ARE KITTY HAWK CLASS SHIPS. To do this, the language may be extended by the call

```
PD[<SHIP>
   (<CLASS> CLASS SHIP)
   (LIST (LIST 'CLASS 'EQ <CLASS>))],
```

where <CLASS> is defined to match class names and takes their data base designations as values.* After this extension, the system will accept such inputs as

```
PRINT THE LENGTH OF KITTY HAWK CLASS SHIPS.
```

A <SHIP> might also match a general category such as CARRIERS, CRUISERS, or MERCHANT SHIPS. Such categories may usually be defined in terms of the TYPE field in the data base. For example, CARRIERS are of type CVA, CVAN, or CVS. OILERS are AO or AOR. <CATEGORY> might be defined by

----

* To simplify the language definition, a language builder may supply LIFER with a preprocessor that does certain kinds of morphological transformations. For example, plural nouns such as SHIPS may be converted to the singular SHIP plus the pluralizing suffix -S. Or, as is assumed here, the suffix may simply be discarded.

```
MS[<CATEGORY>
   ((CARRIER . ((TYPE EQ CV)
                OR (TYPE EQ CVAN)
                OR (TYPE EQ CVS)))
    (OILER . ((TYPE EQ AO)
             OR (TYPE EQ AOR)))
    etc.)].
```

A new production for <SHIP> may then be added such as

```
PD [<SHIP>
    (<CATEGORY>)
    (LIST <CATEGORY>)].
```

With this production, the command

```
PRINT THE LOCATION OF CARRIERS
```

will be accepted.

Modifiers such as AMERICAN, NUCLEAR, and CONVENTIONAL are also very useful; for example,

```
MS[<MOD>
   ((AMERICAN . (NATION EQ US))
    (NUCLEAR . (FUEL EQ NUCLEAR))
    (CONVENTIONAL . (FUEL EQ DIESEL))
    etc.)].
```

By adding

```
PD[<SHIP>
   (<MOD> <SHIP>)
   (CONS <MOD> <SHIP>)],
```

the system will then process inputs such as

```
GIVE ME THE POSITION OF THE AMERICAN NUCLEAR CARRIERS.
```

Superlative modifiers, such as FASTEST and SHORTEST, may be defined:

```
MS[<MOD>
   ((FASTEST . (* MAX SPEED))
    (SLOWEST . (* MIN SPEED))
    (LONGEST . (* MAX LENGTH))
    etc.)].
```

Then the system will accept inputs such as

```
GIVE ME THE NAME AND LOCATION OF THE FASTEST AMERICAN OILERS.
```

This would translate into the IDA call

```
(IDA '((* MAX SPEED) (NATION EQ US)
       ((TYPE EQ AO) OR (TYPE EQ AOR))
       (? NAM) (? LOC))).
```

b. Generalizing <L.T.G>

New sentence-level productions, defined in terms of the more primitive metasymbols described to LIFER above, also greatly extend the range of language accepted. For example,

```
PD[<L.T.G>
   (<PRESENT> <SHIP>)
   (IDA (CONS '(? NAM) <SHIP>))]
```

allows inputs such as

```
    WHAT ARE THE FASTEST NUCLEAR SUBMARINES
    PRINT THE CARRIERS
and GIVE THE KITTY HAWK CLASS SHIPS.
```

As another example,

```
PD[<L.T.G>
   (WHO COMMANDS THE <SHIP>)
   (IDA (CONS '(? COMMANDER) <SHIP>))]
```

allows the input

WHO COMMANDS THE KENNEDY?

c. Calculated Answers

Sometimes a data base does not contain the information needed to answer a question directly, but nevertheless contains information that may be used as input to a procedure that can compute the answer from more primitive data. For example, the distance between two ships is not directly available in the example data base, although position data is. Suppose the function STEAM.TIME can take speed and position information returned by IDA and calculate the time for the first ship to travel to the position of the second ship. Then, after defining <SHIP2> like <SHIP>,

```
PD[<SHIP2>
   (<SHIP>)
   <SHIP>],
```

a new top-level production may be defined as follows:

```
PD[<L.T.G>
   (HOW MANY HOURS IS <SHIP> FROM <SHIP2>)
   (STEAM.TIME
     (IDA (APPEND '((? SPEED)(? LOC)) <SHIP>))
     (IDA (CONS '(? LOC) <SHIP2>)))].
```

This production allows such queries as

```
HOW MANY HOURS IS KENNEDY FROM THE CONSTELLATION .
```

5. Extending the Lexicon with Predicates

In certain instances, it is impractical to use the MS function to explicitly list all of the symbols that might match some metasymbol. For example, if the metasymbol <NUMBER> is to match any number, then MS is of little value. For such cases, LIFER allows a metasymbol to be associated with a predicate function. The metasymbol will match any symbol for which the predicate returns a non-NIL value. When such a match occurs, the metasymbol will take as its semantic value the response returned by the application of the predicate.

To define a metasymbol in terms of a predicate, the function MP (Make Predicate) is used. For example,

```
MP [<NUMBER>  NUMBERP]
```

defines <NUMBER> to match any symbol for which LISP predicate function NUMBERP returns a non-NIL value. When applied to numbers, NUMBERP returns the number itself. When applied to anything else, it returns NIL.

As the following questions indicate, <NUMBER> has many uses in the example data base:

```
WHAT CARRIERS HAVE LENGTHS GREATER THAN 1000 FEET
HOW FAR IS CONSTELLATION FROM 40 DEGREES NORTH 6 DEGREES EAST
WHAT SHIPS ARE WITHIN 100 MILES OF KENNEDY.
```

As the size of the lexicon becomes large, the predicate feature may be used to push certain large classes of words out of the natural language system and into the data base itself. For example, <SHIP-NAME> could be defined in terms of a predicate that accesses the NAM field of the data base. (This would slow the parsing operation, of course, and spelling correction could not be performed easily.)

6. Accepting Metalanguage Inputs

a. <u>Interrogating the Language System</u>

It is possible to define input patterns that make reference to the LIFER package itself. For example, LIFER contains a function called SYMBOL.INFO, which takes a metasymbol as its argument and prints lexical items, patterns, and predicates that may be used to match the symbol. The interface builder may incorporate this function in response expressions as in

```
PD[<L.T.G>
   (HOW IS <SYMBOL> USED)
   (SYMBOL.INFO <SYMBOL>)]
```

After this call to PD,* a user might ask the metaquestion

```
                           HOW IS <SHIP> USED
```

and receive the reply

```
<SHIP> MAY BE ANY SEQUENCE OF WORDS FOLLOWING ONE OF THE PATTERNS:
  <SHIP> => <SHIP-NAME>
            THE <SHIP>
            <CLASS> CLASS SHIP
            <MOD> <SHIP> .
```

Using other system interrogation functions, it is possible to provide in an application language for such inputs as

```
PRINT THE GRAMMAR ON FILE APP.GRAM
DISPLAY THE PRODUCTIONS EXPANDING <SHIP>
SHOW LEXICAL ENTRIES FOR CATEGORY <SHIP-NAME>
WHAT PREDICATE DEFINES <NUMBER>
DRAW THE SYNTAX TREE FOR THE LAST INPUT
HOW WOULD YOU PARSE "HOW FAST IS KENNEDY"
IN WHAT PRODUCTIONS DOES <SHIP> APPEAR ON THE RIGHT .
```

Metaquestions requesting general information about the system, such as

```
WHAT KIND OF INFORMATION DO YOU KNOW ABOUT
WHAT'S IN THE DATA BASE
HELP
```

----

* As specified more fully in [26], the metasymbol <SYMBOL> may itself be defined by function MP, using a predicate that sees if its argument is included in the list of defined metasymbols. Being so defined, <SYMBOL> can even match itself so that the input

```
                    HOW IS <SYMBOL> USED
```

may be parsed and answered.

may easily be included in the application language. Top-level response expressions for such inputs may simply return canned explanation texts. With more sophistication, the expressions might access a semantic schema of the data base (as defined in Section IV below) to help formulate an up-to-date reply.

b. Personalizing the Application Language

LIFER contains a function called SYNONYM that allows a new word to be defined as having the same meaning as a model word that is already known to LIFER. Using this function, an interface builder may introduce structures into the application language that allow users to define their own synonyms at run time. In particular,

```
PD[<L.T.G>
   (DEFINE <NEW-WORD> LIKE <OLD-WORD>)
   (SYNONYM <NEW-WORD> <OLD-WORD>)]
```

allows the parser to accept inputs such as

DEFINE JFK LIKE KENNEDY.

The symbols <NEW-WORD> and <OLD-WORD> are defined by predicates that will match any word. SYNONYM works by copying lexical information from <OLD-WORD> to <NEW-WORD>.

LIFER also contains a function called PARAPHRASE that allows a new sequence of words to be defined as having the same meaning as a model sequence of words that the parser already accepts as a complete sentence. Using function PARAPHRASE in a response expression, the interface builder may extend the grammar by

```
PD[<L.T.G>
   (LET <NEW-SEQUENCE> BE A PARAPHRASE OF <OLD-SENTENCE>)
   (PARAPHRASE <NEW-SEQUENCE> <OLD-SENTENCE>)] ,
```

where <NEW-SEQUENCE> matches any sequence of words and returns a list of the matched words as its value, and <OLD-SENTENCE> matches any sequence of words currently accepted as a sentence in the application language.

This new rule allows computer-naive users to personalize the syntactic constructions understood by the system at run time. For example, the user might say

```
LET "REPORT ON KENNEDY" BE A PARAPHRASE OF
    "PRINT THE LOCATION AND COMMANDER OF KENNEDY" .
```

The expression associated with the top-level production that matches this input sentence calls upon the paraphraser. Given the language definition defined above, LIFER then automatically adds the new production

```
<L.T.G>  =>  REPORT ON <SHIP>
```

to the system, with an appropriate response expression. This new, user-defined production will allow the system to accept such new inputs as

```
REPORT ON THE KENNEDY
REPORT ON OILERS
REPORT ON THE FASTEST AMERICAN SUBMARINES.
```

LIFER's methods for learning paraphrases are discussed below.

7. Extendability

The subsections above have indicated how a few simple notions may be drawn together to create a small interface. But can the same notions be used to create much more sophisticated systems? Until our recent experience, we would have joined others in answering, "Not likely." Long before reaching an acceptable level of performance, previous language systems, including our own, have generally grown so complex and unwieldy that further extension has been stifled.

In designing LIFER, much attention has been given to the problem of supplying interface builders with an environment supporting the incremental development of relatively broad interfaces. All LIFER functions are interactive. Parsing and language specification tasks may be intermixed, allowing interface builders to operate in a rapid, extend-and-test mode. Transition trees, which are an efficient representation for the parser to work with, are automatically produced from productions, which we have found to be an efficient representation for interface builders to work with. The system contains a grammar editor and numerous special functions for answering questions about the structure of the language definition and for tracing and debugging a grammar. Details concerning these and other features of LIFER are specified more fully in the LIFER Manual [26].

We believe that the support features of LIFER have enabled us to give the INLAND language broader coverage than previous systems. Unfortunately, we know of no adequate measure of "breadth of coverage." However, some feeling for the types of inputs accepted by LADDER may be gained by considering a sample of acceptable inputs, such as that shown previously in Table 1.

B. THE TRANSITION TREE PARSER

The LIFER parser is a top-down, left-to-right parser based on a simplification of Woods' [49] augmented transition network (ATN) system. Rather than use true ATNs, LIFER works with transition trees. If <L.T.G> is defined by the productions

```
<L.T.G> => <PRESENT> THE <ATTRIBUTE> OF <SHIP> ¦ e1
        => <PRESENT> <SHIP'S> <ATTRIBUTE> ¦ e2
        => HOW MANY <SHIP> ARE THERE ¦ e3
        => HOW MANY <SHIP> ARE THERE WITH <PROPERTY> ¦ e4
```

then the transition (not syntax) tree of Figure 5 would be constructed for use by the parser. Starting at the box labeled <L.T.G>, the parser attempts (nondeterministically) to move toward the response expressions on the right. At each step, the parser may move to the right on a branch if the left part of the remaining portion of the input can be matched by the symbol on the branch. Literal words on a branch can be matched only by themselves. A metasymbol, such as <PRESENT>, may be matched by a lexical item in the associated set created by MS. Or it may be matched by the predicate, if any, that has been defined for the metasymbol. Or it may be matched by successfully transversing some branch of the transition tree that encodes the productions expanding the metasymbol.

At the top level, if the parser reaches a response expression as a result of accounting for the last word of an input, then a top-level match for the input has been found and the response expression is evaluated to compute a response.

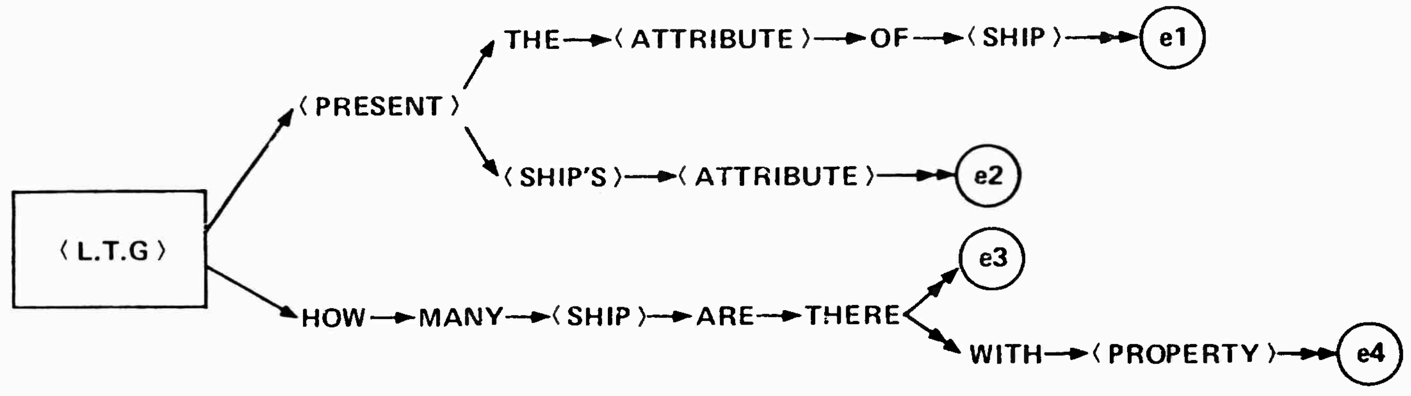

⟨ L.T.G ⟩
⟨ PRESENT ⟩
THE → ⟨ ATTRIBUTE ⟩ → OF → ⟨ SHIP ⟩ → e1
⟨ SHIP'S ⟩ → ⟨ ATTRIBUTE ⟩ → e2
HOW → MANY → ⟨ SHIP ⟩ → ARE → THERE
e3
WITH → ⟨ PROPERTY ⟩ → e4

FIGURE 5 A TRANSITION TREE

C. IMPLEMENTATION OF SPECIAL LIFER FEATURES

This section presents an overview of LIFER's implementation of the spelling corrector, elliptical processor, and paraphraser.

1. Implementation of Spelling Correction

Each time LIFER's left-to-right, ATN parser discovers that it can no longer follow transitions along the current path, it records the failure on a failpoint list. Each entry on this list indicates the state of the system when the failure occurred (i.e., the position in the transition net and the values of various stacks and registers) and the current position in the input string. Local ambiguities and false paths make it quite normal for many failpoints to be noted even when a perfectly acceptable input is processed.

If a complete parse is found for an input, the failpoints are ignored. But if an input cannot be parsed, the list of failpoints is used by the spelling corrector, which selects those failpoints associated with the rightmost position in the input at which failpoints were recorded. It is assumed that failpoints occurring to the left were not caused by spelling errors, since some transitions using the words at those positions must have been successful because there are failpoints to their right.*

The spelling corrector further restricts the rightmost failpoints by looking for cases in which a rightmost failpoint G is dominated by another rightmost failpoint F. G is dominated by F if G is a failpoint at the beginning of a subordinate transition tree that was reached in an attempt to expand F.

Working with the rightmost, dominating failpoints, the spelling corrector finds all categories of words that would be valid at the point

----

* This heuristic can cause LIFER to fail to find and correct certain errors. For example, if the user types CRAFT for DRAFT in WHAT DRAFT DOES THE ROARK HAVE, the spelling error will not be caught since a sentence such as WHAT CRAFT ARE NEAR ROARK would account for the initial sequence WHAT CRAFT. This is traded off against faster processing for the majority of spelling errors.

where the suspected misspelling occurred. This typically requires an exploration of subgrammars. Using the INTERLISP spelling corrector, the word of the input string associated with the rightmost failpoints is compared with the words of the categories just found. If the "misspelled" word is sufficiently similar to any of these lexical items, the closest match is substituted. Failpoints associated with lexical categories that include the new word are then sequentially restarted until one leads to a successful parse. (This may produce more spelling correction farther to the right.) If all restarts with the new word fail, other close lexical items are substituted for the "misspelled" word. If these also fail, LIFER prints an error message.

2. Implementation of Ellipsis

LIFER's mechanism for treating elliptical inputs presumes that the application language is defined by a semantic grammar so that a considerable amount of semantic information is encoded in the syntactic categories. Thus, similar syntactic constructions are expected to be similar semantically. LIFER's treatment of ellipsis is based on this notion of similarity. During elliptical processing, LIFER is prepared to accept any string of words that is syntactically analogous to any contiguous substring of words in the last input. (If the last input was elliptical, its expansion into a complete sentence is used.)

LIFER's concept of analogy appeals to the syntax tree of the last input that was successfully analyzed by the system. For any contiguous substring of words in the last input, an "analogy pattern" may be defined by an abstraction process that works backward through the old syntax tree from the words of the substring toward the root. Whenever the syntax tree shows a portion of the substring to be a complete expansion of a syntactic category, the category name is substituted for that portion. The analogy pattern is the final result after all such substitutions.

For example, consider how an analogy pattern may be found for the substring

OF SANTA INEZ,

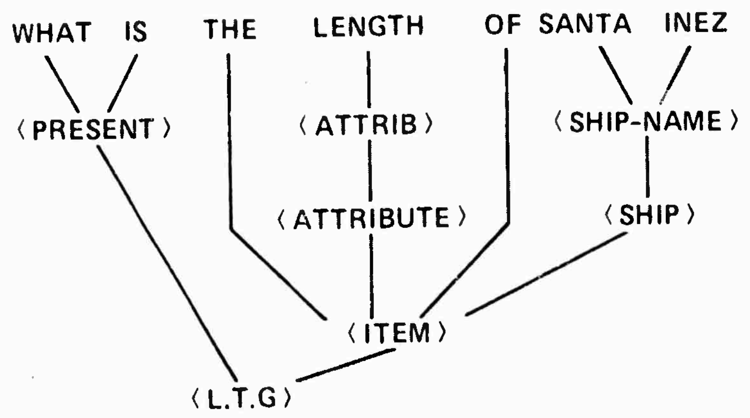


FIGURE 6 A SYNTAX TREE

using the syntax tree shown in Figure 6 for a previous input, WHAT IS THE LENGTH OF SANTA INEZ? Note that the syntax tree used here reflects production rules similar to those defined previously, but introduces a new metasymbol, <ITEM>, to add more substance to the discussion. Since the SANTA INEZ portion of the substring is a complete expansion of <SHIP-NAME>, the substring is rewritten as OF <SHIP-NAME>. Similarly, since <SHIP> expands to <SHIP-NAME>, the substring is rewritten as OF <SHIP>. Since no other portions of the substring are complete expansions of other syntactic categories in the tree, the process stops and OF <SHIP> is accepted as the most general analogy pattern. If the current input matches this analogy pattern, LIFER will accept it as a legitimate elliptical input. For example, the analogy pattern OF <SHIP>, extracted from the last input, may be used to match such current elliptical inputs as

```
    OF THE KENNEDY
    OF THE FASTEST NUCLEAR CARRIER
and OF KITTY HAWK CLASS SHIPS .
```

Note that the expansion of <SHIP> need not parallel its expansion in the old input that originated the analogy pattern. For example, OF KITTY HAWK CLASS SHIPS is not matched by expanding <SHIP> to <SHIP-NAME> but by expanding <SHIP> to <CLASS> CLASS SHIP.

To compute responses for elliptical inputs matching OF <SHIP>, LIFER works its way back through the old syntax tree from the common parent of OF <SHIP> toward the root. First, the routine for computing the value of an <ITEM> from constituents of the production

<ITEM> => THE <ATTRIBUTE>

is invoked, using the new value of <SHIP> (which appeared in the current elliptical input) and the old value of <ATTRIBUTE> from the last sentence. Then, using the newly computed value for <ITEM> and the old value for <PRESENT>, a new value is similarly computed for <L.T.G>, the root of the syntax tree.

Some other substrings with their associated analogy patterns are shown below, along with possible new elliptical inputs matching the patterns.

substring:	THE LENGTH
pattern:	THE <ATTRIBUTE>
a match:	THE BEAM AND DRAFT
substring:	LENGTH OF SANTA INEZ
pattern:	<ATTRIBUTE> OF <SHIP>
a match:	HOME PORTS OF AMERICAN CARRIERS
substring:	WHAT IS THE LENGTH
pattern:	<PRESENT> THE <ATTRIBUTE>
a match:	PRINT THE NATIONALITY
substring:	WHAT IS THE LENGTH OF SANTA INEZ
pattern:	<L.T.G>
a match:	[any complete sentence]

For purposes of efficiency, LIFER's elliptical routines have been coded in such a way that the actual generation of analogy patterns is avoided.* Nevertheless, the effect is conceptually equivalent to attempting parses based on the analogy patterns of each of the contiguous substrings of the last input.

----
* See Hendrix [23] for details of the algorithm.

3. Implementation of Paraphrase

LIFER's paraphrase mechanism also takes advantage of semantically oriented syntactic categories and makes use of syntax trees. In the typical case, the paraphraser is given a model sentence, which the system can already understand, and a paraphrase. The paraphraser's general strategy is to analyze the model sentence and then look for similar structures in the paraphrase string.

In particular, the paraphraser invokes the parser to produce a syntax tree of the model. Using this tree, the paraphraser determines all proper subphrases of the model, i.e., all substrings that are complete expansions of one of the syntactic categories listed in the tree. Any of these model subphrases that also appear in the paraphrase string are assumed to play the same role in the paraphrase as in the model itself. Thus, the semantically oriented syntactic categories that account for these subphrases in the model are reused to account for the corresponding subphrases of the paraphrase. Moreover, the relationship between the syntactic categories that is expressed in the syntax tree of the model forms a basis for establishing the relationship between the corresponding syntactic units inferred for the paraphrase.

i. Defining a Paraphrase Production

To find correspondences between the model and the paraphrase, the subphrases of the model are first sorted. Longer phrases have preference over shorter phrases; for two phrases of the same length, the leftmost is taken first. For example, the sorted phrases for the tree of Figure 6 are:

1.	<ITEM>	THE LENGTH OF SANTA INEZ	
2.	<PRESENT>	WHAT IS	
3.	<SHIP-NAME>	SANTA INEZ	--not used
4.	<SHIP>	SANTA INEZ	
5.	<ATTRIB>	LENGTH	--not used
6.	<ATTRIBUTE>	LENGTH .	

Because the syntax tree indicates <SHIP> => <SHIP-NAME> => SANTA INEZ, both <SHIP-NAME> and <SHIP> account for the same subphrase. For such cases, only the most general syntactic category (<SHIP>) is considered. The category <ATTRIB> is similarly dropped.

Beginning with the first (longest) subphrase. the subphrases are matched against sequences of words in the paraphrase string. (If a subphrase matches two sequences of words, only the leftmost match is used.) The longer subphrases are given preference since matches for them will lead to generalizations incorporating matches for the shorter phrases contained within them. Whenever a match is found, the syntactic category associated with the subphrase is substituted for the matching word sequence in the paraphrase. This process continues until matches have been attempted for all subphrases.

For example, suppose the paraphrase proposed for the question of Figure 6 is

FOR SANTA INEZ GIVE ME THE LENGTH .

Subphrases 1 and 2, listed above, do not match substrings in this paraphrase. Subphrase 3 is not considered, since it is dominated by subphrase 4. Subphrase 4 does match a sequence of words in the paraphrase string. Substituting the associated category name for the word sequence yields a new paraphrase string:

FOR <SHIP> GIVE ME THE LENGTH .

Subphrase 5 is not considered, but subphrase 6 matches a sequence of words in the updated paraphrase string. The associated substitution yields

FOR <SHIP> GIVE ME THE <ATTRIBUTE> .

Since there are no more subphrases to try, the structure

<L.T.G> => FOR <SHIP> GIVE ME THE <ATTRIBUTE>

is created as a new production to account for the paraphrase and for similar inputs, such as

FOR THE FASTEST AMERICAN SUB GIVE ME THE POSITION AND HOME PORT .

ii. Defining a Response Expression for the Paraphrase Production

A new semantic response expression indicating how to respond to inputs matching this paraphrase production is programmed automatically from information in the syntax tree of the model. In particular, the syntax tree indicates which productions were used in the model to expand various syntactic categories. Associated with each of these productions is the corresponding response expression for computing the interpretation of the subphrase from subphrase constituents. The paraphraser reuses selected response expressions of the model to create a new expression for the paraphrase production. The evaluation of this new expression produces the same effect that would be produced if the expressions of the model were reevaluated. Metasymbols that appear in both the paraphrase production and the model remain as variables in the new response expression. Those symbols of the model that do not appear in the paraphrase production are replaced in the expression by the the constant values to which they were assigned in the model.

D. DISCUSSION

As implied by Table 1 and the examples of Appendix 1, the INLAND system is a habitable, rather robust, real-time interface to a large data base and is fully capable of successfully accepting natural language inputs from inexperienced users. In the sections above, we have indicated some of the key techniques used in creating this system. We now seek to place our previous remarks in perspective by considering some of the limitations of the system, the roles played by the nature of our task and the tools we built in developing the system, and some of the similarities and differences between other systems and our own.

1. Limitations

In considering the limitations of our work, the reader should distinguish between limitations in the current INLAND grammar and limitations in the underlying LIFER system.

a. Syntactic Limitations

i. The Class of Languages Covered by LIFER

Consider the set of sentences that LIFER can accept. Because, in the worst case, a special top-level production may be defined in LIFER to cover any (finite-length) sentence that an interface builder may wish to include in the application language, it is impossible to exhibit a single sentence that the LIFER parser cannot be made to accept. Therefore, the only meaningful questions concerning syntactic limitations of LIFER must relate to LIFER's ability to use limited memory in covering infinite or large finite sets of sentences.

LIFER application languages are specified by augmented context-free* grammars. Each rule in the grammar, as discussed previously, includes a context-free production, plus an arbitrarily complex response expression, which is the "augmentation." Although a purely context-free system would severely restrict the set of (non-finite) languages that LIFER could accept, the use of augmentation gives the LIFER parser the power of a Turing machine. The critical question is whether the context-free productions and their more powerful augmentations can be made to support one another in meaningful ways.

To see the interplay between augmentation and context-free rules in the recognition of a classic example of non-context-free languages, consider the language composed of one or more X's followed by an equal number of Y's followed by an equal number of Z's. Let <x> be defined as

```
<x>  =>  X      | 1
<x>  =>  X <x>  | (PLUS 1 <x>)
```

Thus, <x> matches an arbitrary sequences of X's and takes as its value the number of X's in the string. Similar definitions may be made for <y> and <z>. A top-level sentence may be defined by the pattern <x><y><z>, but the augmentation must check to see that the numeric values assigned to the metasymbols are all equal. If they are equal,

----

* See Hopcroft and Ullman [27] for definitions of terms such as "context-free" and "context-sensitive."

the augmentation expression returns some appropriate response. But if they are unequal, the expression returns the special symbol *ERROR*, which the LIFER parser traps as a "semantic" (as opposed to syntactic) rejection.

The Turing machine power of LIFER is illustrated by the following trivial grammar:

```
<PRE-SENTENCE> => <WORD> ¦ (LIST <WORD>)

               => <WORD> <PRE-SENTENCE>
                         ¦ (CONS <WORD>
                                 <PRE-SENTENCE>)

<SENTENCE>     => <PRE-SENTENCE>
                         ¦ (TMPARSE <PRE-SENTENCE>)  .
```

This grammar simply collects all the words of the input into a list, which is then passed to function TMPARSE, a parser of Turing machine power. In this extreme case, the LIFER parser makes virtually no use of the context-free productions, but relies exclusively on the augmentation. LIFER is best used in the middle ground between this extreme and a purely context-free system.

In other words, the class of languages for which LIFER was designed may be characterized as those allowing much of their structure to be defined by context-free rules but requiring occasional augmentation. It has been our experience that much of the subset of English used for asking questions about a command and control data base falls in this class. However, we have not considered certain complex types of transformations which will be discussed in the next subsection.

ii. <u>Troublesome Syntactic Phenomena</u>

English speakers and writers often omit from a sentence a series of words that do not form a complete syntactic unit. For example, consider the following family of conjunctive sentences:

(1) WHAT LAFAYETTE AND WASHINGTON CLASS SUBS ARE WITHIN 500 MILES OF GIBRALTAR

(2) WHAT LAFAYETTE CLASS AND WASHINGTON CLASS SUBS ARE WITHIN 500 MILES OF GIBRALTAR

(3) WHAT LAFAYETTE CLASS SUBS AND KITTY HAWK CLASS CARRIERS IN THE ATLANTIC ARE WITHIN 500 MILES OF GIBRALTAR

(4) WHAT LAFAYETTE CLASS SUBS IN AND PORTS ON THE ATLANTIC ARE WITHIN 500 MILES OF GIBRALTAR

(5) WHAT LAFAYETTE CLASS SUBS IN THE ATLANTIC AND KITTY HAWK CLASS CARRIERS IN THE MEDITERRANEAN SOON WILL BE WITHIN 500 MILES OF GIBRALTAR

Sentence 1 omits the fragment CLASS SUBS ARE WITHIN 500 MILES OF GIBRALTAR from the "complete" question WHAT LAFAYETTE CLASS SUBS ARE WITHIN 500 MILES OF GIBRALTAR AND WHAT WASHINGTON CLASS SUBS ARE WITHIN 500 MILES OF GIBRALTAR. Note that the omitted fragment does not correspond to any well-formed syntactic unit, but begins in the middle of the noun phrase WHAT LAFAYETTE CLASS SUBS and continues to its right. Moreover, the fragment of the noun phrase that is left behind, namely WHAT LAFAYETTE, is not likely to be a well-formed syntactic unit, because one would expect to have WHAT combine with LAFAYETTE-CLASS-SUBS rather than have WHAT-LAFAYETTE combine with CLASS-SUBS. As the family of sentences above illustrates, the omission of words, signalled by the conjunction AND, may be moved to the right through the sentence one word at a time, slicing up the well-formed syntactic units at arbitrary positions. INLAND has no difficulty in accepting either conjunctions or disjunctions of well-formed syntactic categories, but LIFER provides no general mechanism for dealing with omissions that slice through categories at arbitrary points.

In the SYSCONJ facility of Woods [50], special mechanisms for handling a large (but not exhaustive) class of conjunction constructions were built into the parser. Roughly, when SYSCONJ encounters the conjunction "and" in an input X-and-Y, it nondeterministically attempts to break both X and Y into three (possibly empty) parts X1-X2-X3 and Y1-Y2-Y3, such that X1-X2-X3-Y2-Y3 and X1-X2-Y1-Y2-Y3 parse as sentences with the same basic syntactic structure. In particular, X2-X3-Y2 and X2-Y1-Y2 must be expansions of the same metasymbol. The effect of SYSCONJ is to transform X1-X2-X3-and-Y1-Y2-Y3 into X1-X2-X3-Y2-Y3 and X1-X2-Y1-Y2-Y3.

For example,

WHAT LAFAYETTE AND WASHINGTON CLASS SUBS ARE THERE

may be analyzed as

```
WHAT empty LAFAYETTE AND WASHINGTON CLASS SUBS ARE THERE
---- ----- --------- --- ---------- ---------- ---------
 X1   X2      X3     and     Y1         Y2        Y3
```

Both X1-X2-X3-Y2-Y3 (WHAT LAFAYETTE CLASS SUBS ARE THERE) and X1-X2-Y1-Y2-Y3 (WHAT WASHINGTON CLASS SUBS ARE THERE) are parsed by what would correspond in INLAND to the sentence-level production

<L.T.G> => WHAT <SHIP> ARE THERE ,

and both X2-X3-Y2 (LAFAYETTE CLASS SUBS) and X2-Y1-Y2 (WASHINGTON CLASS SUBS) are expansions of the same metasymbol, <SHIP>. In effect, the original input is transformed into WHAT LAFAYETTE CLASS SUBS ARE THERE and WHAT WASHINGTON CLASS SUBS ARE THERE.

Handling conjunctions is just one example of the general need to perform transformations at parse time. A similar phenomenon occurs with comparative clauses, but much more is omitted and transformed. For example,

THE KITTY HAWK CARRIES MORE MEN THAN THE WASHINGTON

may be viewed as a transformed and condensed form of

THE KITTY HAWK CARRIES X-MANY MEN AND
THE WASHINGTON CARRIES Y-MANY MEN AND
X IS MORE THAN Y.

For further discussion of this subject, see Paxton [35].

iii. <u>YES/NO Questions</u>

A limitation of INLAND, although not of LIFER, is that few YES/NO questions are covered. The reason for this is pragmatic: INLAND users do not ask them. Upon reflection, their motivation is clear -- WH questions (i.e., questions asking who, what, where, when, or how) produce more information for the questioner at a lower cost. A user might ask

IS THE KENNEDY 1000 FEET LONG

but it is shorter to ask

HOW LONG IS THE KENNEDY ,

and if the answer to the first question is NO (and if the system is so inconsiderate that it does not indicate the correct length), then the user may still have to ask for the length.

Creating a grammar for YES/NO questions is easy enough. For example,

```
PD[<L.T.G>
   (IS <NUMBER> <UNIT> THE <ATTRIB> OF <SHIP>)
   (YESNO.NUM.ATT <NUMBER>
     <UNIT> <ATTRIB> <SHIP>)]
```

might be used to allow the input

IS 1000 FEET THE LENGTH OF THE KENNEDY.

Function YESNO.NUM.ATT finds the <ATTRIB> of <SHIP> using IDA. Knowing the units in which the data base stores values of <ATTRIB>, YESNO.NUM.ATT converts the returned answer into the units specified by <UNIT> and compares the converted value to <NUMBER>. If the units are valid and the numbers match, YES is returned; otherwise NO is returned and the correct answer, as computed by IDA, is printed.

iv. Assertions

INLAND was designed for retrieval and therefore does rot handle such inputs as

```
THE LENGTH OF THE KENNEDY IS 1072 FEET.
LET THE LENGTH OF THE KENNEDY BE 1072 FEET.
SET THE LENGTH OF THE KENNEDY TO 1072 FEET.
```

Moreover, IDA itself does not provide for updating the data base. Extending the language with new productions such as

<L.T.G> => SET THE <ATTRIBUTE> OF <SHIP> TO <VALUE>

would be easy, but there are serious data base issues involved regarding consistency, security, and priority. Such data base problems are beyond the scope of our research.

v. Irregular Coverage

One consequence of the ease with which interface builders can add new patterns to a LIFER grammar is that gaps may appear in coverage. For example, suppose a given language definition contains no passive constructions. Through the use of paraphrase or by direct action on the part of the interface builder, the language may be extended to cover some, but perhaps not all, passive constructions. That is, the system might be made to accept

1) THE KENNEDY IS OWNED BY WHOM

but not

2) THE KENNEDY IS COMMANDED BY WHOM.

(The semantically oriented syntactic categories for OWNED and COMMANDED may differ.) If a user knows that the system accepts question 1 and that the system accepts the active

3) WHO COMMANDS THE KENNEDY

then he is likely to be upset when input 2 is not accepted.

In creating the language specification for INLAND, we have tried to minimize such irregularities in coverage by applying standard techniques of modular programming to the grammar specification. This, we feel, has been reasonably successful. Because LIFER gives the inference builder the freedom to add particular instances and subclasses of linguistic phenomena, it is his responsibility to avoid the gaps in coverage that may result.

b. Limitations Regarding Ambiguity

The LIFER parser does not deal with syntactic ambiguity directly, but accepts its first successful analysis as being the sole interpretation of an input.* Because English contains truly ambiguous constructions, even when semantic considerations (the "augmentations") are taken into account, this limitation can be serious. For example, in the request

(1) NAME THE SHIPS FROM AMERICAN HOME PORTS THAT ARE WITHIN 500 MILES OF NORFOLK

the phrase THAT ARE WITHIN 500 MILES OF NORFOLK might modify either the SHIPS or the PORTS. The choice will, of course, influence the response made to the user. The current LIFER parser is biased against deep parses and will consider only the interpretation in which the clause modifies SHIPS. Even a single word can produce difficulties. For example, the word NORFOLK in request 1 could refer to a port in Virginia, a port in Great Britain, an American frigate, or a British destroyer. Thus, the request is at least eight ways ambiguous.

Codd [12] has studied at some length the problem of ambiguity in the context of practical data base systems and has developed the strategy of engaging in a dialog in which the system articulates ambiguities (and other problems) and asks questions of the user to clarify the intent of his requests.

In addition to the simple syntactic form of ambiguity, exemplified by request 1 above, other forms of ambiguity may arise. For example, the question

2) IS KENNEDY IN RADAR RANGE OF THE KNOX

is syntactically unambiguous, but the meaning might be either

IS KENNEDY IN KNOX'S RADAR RANGE
or IS KNOX IN KENNEDY'S RADAR RANGE.

This example represents a purely semantic ambiguity.

----
* UPDATE: On October 31, 1977, LIFER was modified to allow optional production of all syntactically correct readings of an input. However, INLAND has not yet been revised to take advantage of this new option. When ambiguity is discovered, LIFER calls a user-defined subroutine with the list of parse trees (including response expressions and variable bindings at each nonterminal node) for all readings. One of the trees is to be returned for execution of the root-level response expression. A default "user" subroutine is supplied with LIFER that prints the various parse trees and asks the user to select one bv number. More sophisticated subroutines are expected to be written that will enter into more natural clarification dialogs.

Similarly,

3) IS KENNEDY NEARER TO GIBRALTAR THAN KITTY HAWK

might be considered syntactically unambiguous in its present form, yet it has two possible meanings. By adding "missing words" at two different points in the input it is possible to produce the readings

```
    IS THE KENNEDY NEARER TO GIBRALTAR THAN
      the kennedy is near to THE KITTY HAWK

and IS THE KENNEDY NEARER TO GIBRALTAR THAN
      THE KITTY HAWK is near to Gibraltar
```

Examples of ambiguity such as queries 1, 2, and 3 begin to show the difficulty of dealing with the problem in any general way. However, in the domain of INLAND, ambiguities have tended to arise only infrequently and have presented only minor problems for our particular application. Fortunately, our users have been very helpful by tending to avoid the use of the long and complex constructions that are most likely to lead to ambiguities. Perhaps this is because the teletype medium inclines users to prefer short, simple constructions.

Even though LIFER does not deal with ambiguity directly, certain types of ambiguities may be trapped and treated by using the response expressions of LIFER production rules. For example,

```
PD[<L.T.G>
   (IS <SHIP1> IN <RANGE-TYPE> RANGE OF <SHIP2>)
   (COMPUTE.RANGE <SHIP1> <SHIP2> <RANGE-TYPE>)]
```

will accept such inputs as

IS KENNEDY WITHIN RADAR RANGE OF KNOX

and call the function COMPUTE.RANGE to respond. COMPUTE.RANGE is given the two ships and the range type as an input. Knowing the pattern to be inherently ambiguous, COMPUTE.RANGE may enter into a (formal) conversation with the user to resolve the ambiguity.

The INLAND grammar also tries to avoid ambiguity whenever possible. For example, the phrase AMERICAN ARMORED TROOP CARRIER might mean a ship (of any nationality) that carries armored troops from the US military, or an American ship that carries armored troops (of any nationality), or a ship that carries troops that were armored by the US, or a ship that was armored by the US and that carries troops, or any one of several other combinations. In INLAND, ARMORED-TROOP-CARRIER is recognized as a fixed phrase and all the problems with ambiguity vanish.

c. Limitations Regarding Definite Noun Phrases

i. The Restricted Context Problem

A phrase such as THE AMERICAN SUBS may be used to refer to different American submarines, depending on the "context" in which it appears. For example, if the Washington, the Churchill, and the Lincoln are being discussed, then THE AMERICAN SUBS in

HOW OLD ARE THE AMERICAN SUBS

refers to the Washington and the Lincoln. Had the current context concerned the Roosevelt, Jefferson, and Leninsky Komsomol, then THE AMERICAN SUBS would have referred to the Roosevelt and the Jefferson. The point is that the meanings of certain noun phrases are dependent upon the contexts in which the phrases appear.

INLAND has only a limited ability to handle phrases such as THE SHIPS, THE AMERICAN SUB, and THOSE CRUISERS, which are said to be "definitely determined." As opposed to indefinitely determined noun phrases (such as A SHIP), which refer to the existence of objects not currently in context, definite noun phrases are often used to refer to a particular object or set of objects that is already in context. In dealing with a data base, "in context" may usually be taken to mean "in the data base." Thus, the phrase THE AMERICAN SUBS generally means "the American subs in the data base," and this is the interpretation that INLAND almost always places on this phrase. But suppose the user has just asked WHAT SUBS ARE IN THE MEDITERRANEAN and has been answered by a list of several subs, some of which are American and some of which belong to other countries. If the user now asks WHAT ARE THE POSITIONS OF THE AMERICAN SUBS, he expects only the positions of American subs in the Mediterranean, but is given information about all American subs in the data base. The problem is that the local context established by previous questions is more restricted than the total data base, and

INLAND has not received enough lexical and syntactic clues to recognize this. (Had the input been WHAT ARE THE POSITIONS OF THE AMERICAN ONES, the use of the pronoun would have signalled the local context and INLAND would have replied properly.)

Where the context is very clear, INLAND can sometimes handle a restricted perspective on the data base. For example, following

SELECT A MAP OF THE NORTH ATLANTIC ,

the query

DISPLAY THE AMERICAN SUBS

will cause the retrieval of only those subs in the North Atlantic, because others could not be displayed on the map in any case.

We know of no applied language system that deals adequately with this problem. However, significant experimental results are described in Grosz [19].

ii. Some Methods for Treating Pronouns

Even though the general problem of properly resolving pronouns is quite difficult, simple techniques can cover a large number of cases. For example, there are many trivial uses of pronouns in which no resolution is needed at all. Examples include

WHAT TIME IS IT
WAS IT 1968 WHEN THE KENNEDY WAS LAUNCHED

and instances in which the pronoun references an earlier phrase in a pattern as in

WHEN WILL <SHIP> <HAVE> ITS <PART> <REPAIRED>
(e.g., WHEN WILL THE KENNEDY GET ITS RADAR FIXED) .

Very often pronouns are used in natural language queries to refer to things mentioned in the previous question. Thus, in the sequence

WHAT IS THE LENGTH OF THE KENNEDY
WHAT IS HER SPEED

the pronoun HER refers to THE KENNEDY. Suppose the first sentence above is interpreted by means of the production

<L.T.G> => <PRESENT> THE <ATTRIBUTE> OF <SHIP>

and the second sentence by

```
<L.T.G>  =>  <PRESENT> <SHIP'S> <ATTRIBUTE> .
```

The primary method for matching <SHIP'S> might be through a production such as

```
<SHIP'S>  =>  <SHIP>  <-'S>
```

where <-'S> is the possessive forming suffix, which is stripped off by a preprocessor.* This primary method may be extended so that <SHIP'S> may also match HER or ITS or THEIR if a <SHIP> was used in the last input. This will allow WHAT IS HER SPEED to be interpreted as WHAT IS KENNEDY'S SPEED.

To extend the definition of <SHIP'S> to match pronouns, first define a predicate SHIP.PRONOUN that will return a non-NIL value if its argument is a possessive pronoun and tne last input contained a <SHIP>. The predicate may be defined as

```
(LAMBDA (WORD)
        (AND (MEMBER WORD '(HER ITS THEIR))
             (LIFER.BINDING '<SHIP>)))
```

where LIFER.BINDING is a LIFER function that determines whether the metasymbol given as its argument had a binding in the interpretation of the last input and, if so, returns the binding.** Using predicate SHIP.PRONOUN, the definition of metasymbol <SHIP'S> may be extended by the call

```
MP(<SHIP'S>  SHIP.PRONOUN) .
```

Another technique, which works nicely for some classes of anaphoric references, involves the use of global variables (sometimes called "registers"). For example, suppose that each response expression associated with a pattern defining the metasymbol <SHIP> is so constructed that it will set the global variable LATEST-SHIP to the value it returns as the binding of <SHIP>. To be concrete,

```
PD[<SHIP>
   (<SHIP-NAME>)
```

----

* Alternatively, a set of possessive nouns naming ships could be defined and the stripper not used, or, as is the case in INLAND, possessives could be thrown away altogether and <SHIP'S> could be made equivalent to <SHIP>.

** If there were multiple occurrences of the symbol in the last input, the leftmost-topmost instance is returned.

```
        (SETQ LATEST-SHIP
              (LIST (LIST 'NAM 'EQ <SHIP-NAME>)))]
```

causes <SHIP> to match a <SHIP-NAME> as defined previously. The response expression that computes the value of <SHIP> will return the same value as defined above, but, as a side effect, it will now also set the global variable LATEST-SHIP to the same value. Later, when phrases such as THE SHIP or THAT SHIP are used to refer to the last ship mentioned, the global variable LATEST-SHIP may be used to recall that ship. For example, if <DET-DEF> is defined to match definite determiners (e.g., THAT, THE), then

```
     PD[<SHIP>
        (<DET-DEF> SHIP)
        LATEST-SHIP]
```

will define structures that allow <SHIP> to match THE SHIP and take as its value the value of LATEST-SHIP. Note that LATEST-SHIP is always ready with the value of the latest <SHIP> mentioned, but (LIFER.BINDING '<SHIP>) is of help only if <SHIP> was used in the last input.

d. Limitations in Processing Elliptical Inputs

After successfully processing the complete sentence

1) HOW MANY CRUISERS ARE THERE?

LIFER will accept the elliptical input

2) CRUISERS WITHIN 600 MILES OF THE KNOX

but not

3) WITHIN 600 MILES OF THE KNOX?

The elliptical processor is based on syntactic analogies. Input 2 is a noun phrase that is analogous to the noun phrase CRUISERS of input 1. Input 3, on the other hand, is a modifier that is intended to modify the CRUISERS of input 1. Because input 1 has no modifiers, elliptical input 3 has no parallel in the original input and hence cannot be accepted.

e. Other Limitations

A few other important limitations of INLAND and LIFER are worth mentioning briefly.

First, LIFER has no "core grammar" that is ready to be used on any arbitrary data base. This is because LIFER was designed as a general purpose language processing system and makes no commitment whatever to the types of programs and data structures for which it is to provide a front end or even to which natural language is to be accepted. LIFER might, for example, be used to build a panese language interface to a program that controls a robot arm. This could not be done if assumptions had been made restricting LIFER to data base applications and to the English language. Thus, LIFER contrasts with systems such as Thompson's REL [43], which provides a core grammar but which requires reformatting of data into the REL data base.*

Some systems, such as ROBOT (Harris [21] [22]), use the information in the data base itself as an extension of the language processor's lexicon. The LIFER interface may do this also but need not. If one elects not to use the data base as lexicon, and this choice was made in INLAND, then the lexicon must be extended whenever new values are added to the data base that a user may want to mention in his queries.** The price of using the data base itself as an extended lexicon is that the data base must be queried during the parsing process. For very large data bases, this operation will probably be prohibitively expensive.

INLAND, of course, is basically a question answerer that relies on a data base as its major source of domain information. In particular, INLAND cannot read newspaper articles or other extended texts and record their meaning for subsequent querying. Moreover, although it is perfectly reasonable that the LIFER parser might be used for a text

----

* Fragments of foreign-language versions of INLAND have been used to access the naval data base in Swedish and Japanese.

** Because LADDER accesses data over the ARPANET, we felt the overall system would be intolerably slow if the actual data base was used at parse time.

reading system, LIFER itself contains no particular facilities other than calls to response expressions for recording or reasoning about complex bodies of knowledge.

2. The Role of the Task Domain

The limitations presented in the last subsection would cause major difficulties in dealing with many areas of natural language application. However, for our particular application, the limitations did not prevent the creation of a robust and useful system. In the next few paragraphs, we briefly outline some of the key features of the application that simplified our task.

The creation of INLAND was greatly facilitated by the nature of the particular interface problem that was addressed -- providing a decision maker with access to information he knows is in a data base. Because the user is expected to know what kinds of information are available and is expected to follow the technical terms and styles of writing that are typical in his domain of decision making, we can establish strong predictions about a user's linguistic behavior so that INLAND needs to cover only a relatively narrow subset of language.

A second factor in facilitating the creation of the natural language interface was the interface provided by the IDA and FAM components of the LADDER system. By providing a simplistic view of what is in fact a complex and highly intertwined collection of distributed data, IDA and FAM helped greatly in simplifying the LISP response expressions associated with productions in the INLAND grammar.

In short, IDA allows the data base to be queried by high-level information requests that take the form of an unordered list of two kinds of items: fields whose values are desired, and conditions on the values of associated fields. Using IDA, the INLAND grammar need never be concerned with any entities in the data base other than fields and field values. Futhermore, because the input to IDA is unordered, the construction of segments of a call to IDA can be done while parsing lower-level metasymbols.

The performance of INLAND for a given user is also enhanced by the user's own, often subconscious, tendency to adapt to the system's limitations. Because INLAND can handle at least the most straightforward paraphrases of most requests for the values of any particular fields, even a new user has a good chance of having his questions successfully answered on the first or second attempt. It has been our experience that those who use the system with some regularity soon adapt the style of their questions to that accepted by the language specification. The performance of these users suggests that they train themselves to understand the grammar accepted by INLAND and to restrict their questions whenever possible to forms within the grammar. Formal investigation of this subjectively observed phenomenon might prove very interesting.

3. The Role of Human Engineering

Although the basic language processing abilities provided by LIFER are similar to those found in some other systems, LIFER embodies a number of human engineering features that greatly enhance its usability. These humanizing features include its ability to deal with incomplete inputs and to allow users to extend the linguistic coverage at run time. But more importantly, LIFER provides easy-to-understand, highly interactive functions for specifying, extending, modifying, and debugging application languages. These features provide a highly supportive environment for the incremental development of sophisticated interfaces. Without these supporting features, a language definition rapidly becomes too complex to manage and is no longer extendable. With support, the relatively simple types of linguistic constructions accepted by LIFER may be used to produce far more sophisticated interfaces than was previously thought possible.

Creating a LIFER grammar that covers the language of a particular application may be thought of constructively as writing a program for a parser machine. All the precepts of good programming -- top-down design, modular programming and the like -- are relevant to good design of a semantic grammar. A well-programmed grammar is easy to augment, because new top-level patterns are likely to refer to lower-level metasymbols that have already been developed and shown to work reliably. Thus, the task of adding new top-level productions to a grammar is analogous to the task of adding new capabilities to a more typical body of computer code (such as a statistics package) by defining new capabilities in terms of existing subroutines.

No matter how well-programmed a grammar might be, as the complexity of the grammar increases, the interactions among components of the languag specification will grow. This leads the language designer into the familiar programming cycle of program, test, and debug. With many systems for parsing and language definition, the cycle may take many minutes for each iteration. With LIFER, when a new production is interactively entered into the grammar, it is immediately usable for testing by parsing sample inputs. The time required for the cycle of program, test, and debug is thus dependent on the thinking time of the designer, not the processing time of the system. Because the designer can make very effective use of his time, he can support, maintain, and extend a language specification of far greater complexity than would otherwise be possible.

The basic parsing technology of LIFER is not really new. But the human engineering that LIFER provides for interface builders has allowed us to better manage the existing technology and to apply it on a relatively large scale.

4. Related Work

As indicated by the February 1977 issue of the SIGART Newsletter [15], which contains a collection of 52 short overviews of various research efforts in the general area, interest in the development of natural language interfaces is widespread. Our own work is similar to that of several others.

The LIFER parser is based on a simplification of the ideas developed in the LUNAR parser of Woods and others [49] [52]. In particular, LIFER manipulates internal structures that reflect Woods' ATN formalism. Woods' parser was used as a component of a system that accessed a data base in answering questions about the chemical analysis of lunar rocks. The system did not use semantically oriented syntactic categories, and the data base was smaller and less complex than that used by INLAND, although the data base query language was more general than that accepted by IDA.

Woods' ATN formalism has been used in a variety of systems, including a speech understanding system [51], and the semantically oriented systems of Waltz [45] [46] and Brown and Burton [6] [7]. These latter systems do not use the LUNAR parser, but rather compile the ATN formalism into procedures that in turn perform the parsing operation directly, without using a parser/interpreter to interpret a grammatical formalism. Compilation results in greater parsing speed, which is of importance for many applications. However, compilation also makes personalization features, such as PARAPHRASE, much more difficult to implement, and increases the time of the program-test-debug cycle.

The first natural language systems to make extensive use of semantic grammars were those of Brown and Burton [6] [7]. These systems are designed for computer-assisted instruction rather than as interfaces to data bases.

In work very similar to our own, Waltz [45] [46] has devised a system called PLANES, which answers questions about the maintenance and flight histories of airplanes. PLANES uses both an ATN and a semantic grammar. Apparently, the system does not include a paraphrase facility similar to LIFER's. It does support the processing of elliptical inputs by a technique differing from ours, and supports clarification dialogues with users.

The PLANES language definition makes less use of syntactic information than INLAND. In particular, PLANES looks through an input for constituent phrases matching certain semantically oriented syntax categories. When one of these constituent phrases is found, its value is placed in a local register that is associated with the given category. Rather than attempt to combine these constituents into a complete sentence by syntactic means, "concept case frames" are used. Essentially, PLANES uses case frames to decide what type of question has been asked by looking at the types and values of local registers that were set by the input. For example, the three questions

```
WHO OWNS THE KENNEDY
BY WHOM IS KENNEDY OWNED
THE KENNEDY IS OWNED BY WHOM
```

would all set, say, an <ACT> register to OWN and a <SHIP> register to KENNEDY. The case frames can determine what question is asked simply by looking at these registers. Performing a complete syntactic analysis such as INLAND does requires different constructions for each question pattern.*

If the input following one of the three questions above is the elliptical fragment KNOX, the <SHIP> register is reset. Because no case frame is associated with <SHIP> alone and because <SHIP> was used in the last input, the <ACT> register is inherited in the new context and the elliptical input properly analyzed. When more than one case frame matches an input, PLANES enters into a clarification dialog with the user to decide which was intended. (This conversation prints interpretations of inputs in a formal query language.)

The use of case frames is very attractive in that it allows many top-level syntactic patterns to be accounted for by a single rule. However, it is inadequate for complex inputs. The question IS KNOX FASTER THAN KENNEDY contains two <SHIP>s. Only the syntax tells us which to test as the faster of the two. Compound-complex sentences

----

* LIFER may be used to support case frames, although this was not done in INLAND. In particular, <L.T.G> may be defined as an arbitrary sequence of <CONSTITUENT>s, where <CONSTITUENT> may be expanded as any of the semantically oriented syntax categories used by the case system. The response expression associated with the expansions of <CONSTITUENT> cause global registers to be set, and the response expression associated with

<L.T.G> => <CONSTITUENTS>

may make use of these registers and the case frames in computing a top-level response. A case frame system supported by LIFER would, of course, inherit LIFER's run-time personalization and introspection features.

would be extremely difficult to process without extensive use of syntactic data. Waltz is investigating ways of supplementing his case frames with nominal pieces of syntactic information.

Codd's concept of the RENDEZVOUS system [12] for interface to relational data bases provides many ideas concerning clarification dialogs that might be included in LIFER at some later date. RENDEZVOUS is fail-safe in that it can fall back on multiple choice selection if natural language processing fails completely.

Another applied natural language system whose underlying philosophy is akin to that of LIFER is the REL (Rapidly Extendable Language) system of Thompson and Thompson [43]. REL is a data retrieval system like LADDER, though REL requires data to be stored in a special REL data base. The grammar rules of REL contain a context-free part and an augmentation very much like those of LIFER. As its name implies, REL was intended to be easily extendable by interface builders. Much effort has gone into making REL run rapidly, and it is almost certainly faster than LIFER. However, this speed was gained by a low-level language implementation with the unfortunate side effect that response expressions are not easily written.

Recently, the Artificial Intelligence Corporation introduced a commercial product called ROBOT for interfacing to data bases. As described in Harris [21] [22], ROBOT "calls for mapping English language questions into a language of data base semantics that is independent of the contents of the data base." The data base itself is used as an extension of the dictionary and the structure of files within the data base helps in guiding the parser in the resolution of ambiguities. Our own research indicates that the types of linguistic construction employed by users are rather dependent on the content of the data base. We also worry that extensive recourse to a data base of substantial size may greatly slow the parsing process, unless the file is indexed on every field. Moreover, our data base is coded largely in terms of abbreviations that are unsuitable as lexical entries. Nevertheless, the notion of using the data itself to extend the capabilities of the language system is very attractive.

In addition to the work on near-term application systems, a number of workers are currently addressing longer-range problems of accessing data bases through natural language. These include Mylopoulos et al. [33], Sowa [40], Walker et al. [44], and ourselves [38]. There are, of course, many people engaged in research in the general area of natural language processing, but a survey of their work is beyond the scope of this report.

E. CONCLUSION

We have described the language interface component of LADDER, which provides natural language access to a large, distributed data base. We have shown that, although it is based on simple principles and subject to certain limitations, it is sufficiently robust to be useful in practical applications. Moreover, we have indicated that LADDER is not an isolated system but that other applied language systems have achieved significant levels of performance as well, particularly in interfacing to data bases. We believe that the evidence presented indicates clearly that, for certain restricted applications, natural language access to data bases has become a practical and practicable reality.

Preceding Page Blank - NOT FILMED

IV INTELLIGENT DATA ACCESS

by Daniel Sagalowicz

A. INTRODUCTION

Although the IDA (for Intelligent Data Access) program was developed as part of the LADDER system, it may also be used independently of the natural language front end.* In fact, as users of a natural query system such as LADDER gain experience with the system, they are likely to desire a more terse (and complete) input language than a subset of English. Thus, LADDER permits IDA to be invoked directly. In this section, we will present IDA as if it were to be used without any language front end. The primary goal of IDA is to provide the user with a "structure-free" interface to a data base. The user of IDA needs to know only field names and, of course, IDA's query language, but is not required to know the data base structure or to employ any other complex structure. The goals of IDA are similar to those of the APPLE system [8], and we refer the reader to that paper for a very clear and complete discussion of the need for such a system. Two other systems that are also intended to free the user from the need to know the details of the data base structure are INGRES [41] and SYSTEM-R [9]. However, neither of these systems provides an interface to the data base that is structure-free. Both allow their users to introduce a new "logical view" of the data base which is different from the actual data base. Since the user can use only the predefined relations that belong to his logical view, he must have available a large set of predefined logical views if he is to access the data in a truly flexible

----

* A formal specification of the direct interface to IDA is provided in Appendix 2.

manner. This means that, to use SYSTEM-R or INGRES, the user must still acquire knowledge of a complex structure -- in this case, the set of various logical views. When using IDA, on the other hand, the user's point of view is that, for each query, the whole data base is configured into only one relation, namely, the relation needed to satisfy his query -- no matter what the query is -- as long as the information is contained in the data base. To obtain the same result using the logical view technique of INGRES, the data base administrator would have to define all logical relations that can be obtained by taking the joins of 2 relations, the joins of 3 relations, and so on. Then the DBMS would have to match the query with all those virtual relations to decide which one applies, and finally execute the query accordingly. In the Blue File data base currently accessed by LADDER, which is composed of 14 relations, some queries require the join of 5 relations. Even with such a small data base, the task of the data base administrator in setting up the logical views that would include all the joins of up to 5 relations would be tremendous. By contrast, IDA requires the data base administrator to define only the pairwise joins among the relations in the data base -- whenever such joins exist. Then, for each particular query, IDA decides dynamically what sequence of joins needs to be performed. In effect, IDA decides dynamically what "logical view" corresponds to the query, and decides dynamically how to satisfy the query in that particular logical view.

As already mentioned, APPLE has goals that are very similar to those of IDA. APPLE attempts to use "paths" to describe all the possible joins and projections allowed in the data base. In APPLE, all the paths have been prestored statically and, at run time, only a choice between those prestored paths is allowed. In IDA, however, the path to be followed is decided entirely at run time; therefore, there is no need for the data base administrator to define all possible paths through the data base.

In the next subsection, we present a simple subset of the Blue File data base, which will be used for the examples. The following subsections discuss IDA's input language, the features of the IDA system, and finally, the primary limitations of the current version of IDA and possible areas for future research.

B. EXAMPLE DATA BASE

For the purposes of this explanation of IDA, we will draw our examples from a simplified version of the Blue File data base that we created on the Datacomputer.

In our examples, we will assume that this data base is relational, with the following relations:

SHIP	:	(NAME, CLASS , TYPE, PTP, PTD, CONVOY)
SHIPCLASS	:	(CLASS, TYPE, LGH, DRAFT)
CONVOY	:	(CONVOY-NAME, ESCORT, PTP, PTD)
UNIT	:	(ANAME, EMBRK, COMMANDER)

where NAME is the name of a ship, CLASS is her class, TYPE is her type (i.e., aircraft carrier), PTP is the ship's last reported position, and PTD the corresponding date. All ships belonging to the same class have the same static characteristics; therefore, LGH is the length of the class, i.e., the length of all ships belonging to that class, DRAFT their draft, and so on. Ships may also belong to convoys, in which case the convoy name is contained in the CONVOY field. The corresponding convoy relation refers to the convoy name in CONVOY-NAME, the escort type in ESCORT, and the date and position of the convoy in PTD and PTP. Finally, the UNIT relation gives the names of units in ANAME, the ship on which they are embarked in EMBRK, their commanding officer's name in COMMANDER.

This simple subset of the Blue File data base will be sufficient to demonstrate IDA's characteristics.

C. IDA'S INPUT LANGUAGE

Because IDA is used in our system with a natural language front end, a great emphasis was put on developing a very simple input language for IDA. Moreover, this simple format makes it very easy to interface any front end to IDA, whether it is a natural language front end, a graphic front end, or a formal query language.

As already mentioned, the main goal of IDA was to relieve the user from needing to know the structure of the data base when issuing an IDA query. A few simple examples will illustrate the format of the input to IDA.

Let us first consider the request:

GIVE THE NAMES OF ALL THE SHIPS.

For this request, the query to IDA would be: (? NAME). In general, to request the value of some field, the user simply precedes the field name by the symbol '?'.

Let us now consider the query:

WHAT IS THE LENGTH OF THE FOX?

In this case, the query to IDA would be:

(? LGH) (NAME EQ 'FOX')

And for the request:

LIST ALL SHIPS WITH LENGTHS BETWEEN 300 AND 1000 FEET,

the query to IDA is:

(? NAME) ((LGH GE 300) AND (LGH LE 1000)).

In general, we may specify some restrictions on field values, using any boolean expression. The comparison operators accepted by the system are: EQ, NE, GT, GE, LT, or LE. In the two questions above, two examples of such boolean restrictions appear: (NAME EQ 'FOX') and ((LGH GE 300) AND (LGH LE 1000)).

Finally, the query:

WHAT IS THE LONGEST SHIP?

would correspond to the IDA query:

(? NAME) (* MAX LGH).

In general, to specify that he is interested in some set of fields in the data base which corresponds to the maximum (or minimum) value of some field, the user simply precedes the corresponding field name by * MAX (or * MIN). In the same way, to find the answer to a "how many?" question such as:

HOW MANY SHIPS ARE LONGER THAN 300 FEET?

the user would query IDA with:

(LGH GE 300) (* COUNT NAME).

This "* feature" was introduced to handle all computations that require an iterative program to be executed by the DBMS. Such is the case with the computations of the maximum, minimum, and count, which are currently implemented. It could easily be extended to other cases such as computations of sums or averages.

Having seen these examples, we can now formally specify the input to IDA: it is a series of lists, each of which may take any one of three different formats:

(? fieldname)

A complex boolean expression

(* <*OP> fieldname) where <*OP> is one of MAX, MIN, or COUNT.

From the above description, it is clear that IDA does not require the user to know the structure of the data base in issuing a query.

D. FUNCTION AND STRUCTURE OF IDA

In this section, we explain the features of IDA, using examples whenever appropriate. To help keep these examples simple and self-explanatory, we will assume that IDA generates calls to a relational data base whose query language is exactly the same as IDA's, except for the explicit presence of relation names. In the system actually developed, the results of IDA are handed to the Datacomputer (via FAM) so that the queries generated by IDA are in Datalanguage, the query language of the Datacomputer. For the interested reader, an actual transcript of a session with our system is included in Appendix 1.

1. IDA's Structural Schema

As indicated above each query to IDA may be considered to be issued against a single relation, the fields of that are all those which appear in the query. IDA must then solve the problem of building this relation dynamically from the actual relations in the data base, using the classical relational operators: projections, restrictions, and joins. To do so, IDA uses a structural model of the data base, called the "structural schema." It is composed of two types of information: "relation frames" and "field frames." These frame structures are similar to those discussed by Minsky [30] and Winograd [48], for example. Each frame is a list of property-value pairs, also called "slots," which provide some specific information about the entity the frame models.

Each relation frame corresponds to an actual relation in the data base, and gives the possible links with all the other relations. In other words, the relation frames define all the permissible joins of two relations. In the case where a direct join is not possible between two specific relations, the two relation frames would instead include the name of a third relation that must be included in the join: in the Blue File data base example, the direct join between the SHIPCLASS and CONVOY relations is not allowed, in which case the join would have to be done among all three relations. As an example, the relation frame for SHIPCLASS would be:

```
          [SHIP          (CLASS)

           CONVOY        {SHIP}]
```

where each slot has the name of a relation and the link with that relation, or the name of a third relation in case of indirect link. In this example, the first slot in the SHIPCLASS relation frame indicates that to join any projections and/or restrictions of the SHIPCLASS and SHIP relations the join must be taken over the CLASS field. The second slot indicates that one is not allowed to take a direct join between the SHIPCLASS and CONVOY relations; one must take the join of the three relations.

Each field frame corresponds to a field name that could appear in a user query. The main information contained in a field frame is the list of all the relations to which this field belongs. In many cases, a field belongs to a single relation: such is the case of LGH, which belongs only to the SHIPCLASS relation in the Blue File data base. In many other cases, a field may belong to several relations: such is the case of TYPE, which belongs to both the SHIP and SHIPCLASS relations of the Blue File data base, and SHIP-NAME, which belongs to both the SHIP relation (under the field name NAME), and the UNIT relation (under the field name EMBRK).

2. IDA's Covering Algorithm

These two types of information -- the links in the relation frames and the relation names in the field frames -- are used by IDA's "covering algorithm" to determine at run time the logical view corresponding to any given user query. More precisely, the role of the covering algorithm is to find the smallest set of relations that cover all the fields in the query -- a set that provides for any two relations, either a legal direct join or an indirect join using only other relations in the set. In other words, information in the relation frames could define a graph in which each node is a relation and the edge between two nodes is the link between the two relations. Then, the role of the covering algorithm is to find the smallest (by the number of nodes) connected subgraph that covers every field in the query. We are currently using a heuristic algorithm that works reasonably well and eliminates the need to do an exhaustive search to find the minimum cover. Although such a search would always be successful, it would be time-consuming, particularly when the number of relations in the data base and the number of fields in the query are large. Our heuristic covering algorithm is described in the paragraph below.

a. IDA's Heuristic Covering Algorithm

At each iteration of the algorithm, we have a list of already chosen relations (empty for the first iteration) and a list of fields in the query not yet covered by the chosen relations. Then, we pick at random one not-yet-covered field and try to find a relation that:

* Covers it.
* Has a direct link with one already chosen relation, if such a relation exists.
* Covers as many not-yet-covered fields as possible, if there is a choice.

The strategy of IDA's covering algorithm is to minimize some cost function. In our current implementation, this cost function is just the number of relations that need to be accessed -- a strategy that is optimal in many, but not all, cases. The strategy is particularly relevant when the relations are on various computers and may al need to be copied on a single computer (this would occur if the DBMS allows, and IDA generates, multi-file queries). However, the strategy would be suboptimal if, for example, the relations were all on the same computer, and if the indexing characteristics of these relations were very different. In this case, a "query cost estimator" of the type developed by Hammer [20] would be needed, and it would indicate the cost that IDA would try to minimize. This may be a worthwhile later addition to the system.

To see how the covering algorithm performs, let us consider the following three examples:

Example 1: WHAT IS THE LENGTH OF BELKNAP CLASS SHIPS?

The corresponding query to IDA is:

```
                    (? LGH) (CLASS EQ 'BELKNAP').
```

The generated program is:

```
          IN SHIPCLASS RELATION: (? LGH) (CLASS EQ 'BELKNAP').
```

Clearly, in this case not much work needs to be done: IDA finds in the structural schema that both CLASS and LGH are in the SHIPCLASS relation, with those very same field names, and issues the corresponding query.

Example 2: LIST THE POSITION OF ALL CARRIERS.

The corresponding IDA query is:

```
                (TYPE EQ 'CV') (? NAME) (? PTP) (? PTD).
```

The generated program is:

```
     IN SHIP RELATION:

        (TYPE EQ 'CV') (? NAME) (? PTP) (? PTD).
```

In this case, we have to go only to the SHIP relation, and not at all to the SHIPCLASS relation. The reason is, of course, that all the fields in the query are covered by the SHIP relation -- even though some are also covered by the SHIPCLASS relation.

Example 3: WHAT IS THE LENGTH OF THE FOX

The query to IDA is:

```
                         (? LGH) (NAME EQ 'FOX').
```

The query program generated is:

```
     IN SHIP RELATION: (NAME EQ 'FOX') (? CLASS)

     IN SHIPCLASS RELATION: (? LGH) (CLASS EQ 'BELKNAP').
```

Here, (CLASS EQ 'BELKNAP') would be filled by the response to the first query. This is a case in which an actual link between relations is required since IDA must perform the join between the SHIP and SHIPCLASS relations. Clearly, several kinds of information are needed to build the correct DBMS queries. First, IDA must determine in which relations the fields are located: this information is found in the field frames, as already indicated. Second, after IDA has decided that two relations need to be accessed, namely, SHIP and SHIPCLASS, it has to determine the "link" between them -- in other words, what kind of information is needed from the first relation to limit the search inside the second relation. In this case, this link is the value of the field CLASS. This linkage information is found in the relation frame for either relation. For example, in the relation frame corresponding to SHIP, one would find a slot corresponding to the SHIPCLASS relation, where the linkage would be indicated: in this case, the field name CLASS would be in this slot.

b. Alias Fields

Besides the frame slots used by the covering algorithm, other slots are defined for the field frames. For example, the frame for NAME is:

```
[RELATIONS (SHIP UNIT)
 ALIAS-IN-UNIT EMBRK],
```

where the first slot corresponds to the list of relations that cover the field, and where the second slot gives the actual name of the field in the UNIT relation. The use of this second slot is illustrated in the following example:

Example 4: ON WHICH SHIP IS VTF164 EMBARKED?

The corresponding query to IDA is:

```
(ANAME EQ 'VTF164') (? NAME).
```

The generated program is:

```
IN UNIT RELATION: (ANAME EQ 'VTF164') (? EMBRK).
```

Although NAME was mentioned in the query to IDA, IDA generated a call using EMBRK instead, which is correct since NAME does not appear in the UNIT relation, and may be replaced by EMBRK. IDA made the replacement because EMBRK was the value in the ALIAS-IN-UNIT slot of the NAME field frame. Note that a different field frame for EMBRK may or may not exist, depending on the intended use of the particular data base. It may therefore be possible to make use of several field frames corresponding to various aliases of the same field. This would allow IDA to handle a query in different ways, depending on which alias of the field name is actually use in the query. In particular, this may be a way to avoid the multi-path problem mentioned by Carlson [8]. Take, for example, a multi-path case that arises even in the simple Blue File data base. Previously, we assumed that the direct join between the SHIP and UNIT relations could only be taken using the link between SHIP and EMBRK; however, it is also possible to join those two relations using the link between SHIP-NAME and ANAME since a ship is also a unit. For example, the question

LIST THE COMMANDING OFFICERS OF ALL AIRCRAFT CARRIERS requires that we take this last join, using the link between the pair NAME in SHIP and ANAME in UNIT. The query to IDA would be:

(? ANAME) (? COMMANDER) (TYPE EQ 'CV').

Since this query would mention ANAME, IDA should assume that the question was restricted to the commanders of units that are aircraft carriers, and therefore should not use the EMBRK field at all. Note that, in this case, we have two field frames that correspond to ship names, namely, ANAME and NAME. Therefore, depending on which field is actually used in the IDA query, the multi-path ambiguity can be eliminated.

c. <u>Special-Purpose Procedures</u>

Although many other slots exist in both the field and the relation frames, and are used for specific purposes by IDA, we will mention here only one final one: the "procedural slot." We will explain its use in the following example.

Example 5: WHERE IS THE TABOR?

The query to IDA would be:

(? PTP) (NAME EQ 'TABOR').

The main problem that is raised by this query is that if the Tabor belongs to a convoy, her position is found in the convoy file -- this is typically done to ease the updating process for ships in convoys. Therefore, IDA must check whether the Tabor does or does not belong to a convoy. In this case, IDA would issue the following query:

IN SHIP RELATION: (NAME EQ 'TABOR') (? CONVOY) (? PTP).

Then, depending on the response, we would issue a susequent query to the convoy relation, or use whatever was the value of PTP in the SHIP relation. This is a typical example of what we call the "conditional case." Somewhere a condition on the value of a field has to be tested, and, depending on the result of the tests, different queries to the data base may be issued.

IDA handles this situation by having in the field frame that corresponds to the PTP field a special program -- a procedural attachment, in Winograd's terms [48]; this program is executed by IDA and does exactly what is needed to handle this case. First, it replaces (? PTP) by (? PTP) (? CONVOY) in the query. When the answer is returned, it checks the answer and either returns the content of PTP or issues an appropriate new query against the CONVOY relation. Procedural attachments of this type may be used not only in the conditional case, but also in cases of complex redundancies. For example, if the total number of ships in a convoy was not stored explicitly in the data base, a special procedural attachment to a SHIP-NUMBER field frame could be used to call IDA to count the number of ships that belong to the convoy.

These five examples have given the reader a reasonably complete picture of the use of the frame slots by IDA. Now we briefly explain how IDA orders the accessing of relations.

d. Relation Ordering

IDA currently decides on the ordering of the relations to access by using two very simple rules:

* IDA tries to defer retrieving the values of the fields requested by the user as long as possible. The intuitive reason for this rule is that acquiring values for fields queried by the user does not provide any additional constraints on any subsequent data base query. All other things being equal, it is best to try to constrain the data base queries as soon as possible by obtaining early those field values to be used in subsequent queries. Since other things are not always equal, this rule is only a "soft" constraint on IDA compared to the next one.
* IDA should not issue a query of the type (* <OP> fieldname) until all other boolean restrictions have been used. This is a strict constraint, since, if not used, IDA's results would be incorrect as Example 7 below will make clear.

We now present two examples of the use of these rules:

Example 6: WHAT IS THE LONGEST SHIP?

The query to IDA is:

(? NAME) (* MAX LGH).

The generated program is:

```
IN SHIPCLASS RELATION: (* MAX LGH) (? CLASS)

IN SHIP RELATION: (? NAME) (CLASS EQ ... ).
```

This example differs from Example 3 only in the order of access to the two relations. In Example 3, the SHIP relation was accessed first since it was the most constrained relation, while in this case the reverse is true.

Example 7: WHAT IS THE LONGEST SHIP BELONGING TO A CONVOY?
The query to IDA is:

```
(? NAME) (CONVOY NE '*') (* MAX LGH).
```

The generated program is:

```
IN SHIP RELATION:
     (? NAME) (CONVOY NE '*') (? CLASS)

IN SHIPCLASS RELATION:
     (* MAX LGH) (CLASS EQ ... OR CLASS EQ ... OR ... ).
```

Clearly, in this case we must not ask to find the class that is the longest until we have found which classes correspond to ships in convoys.

Therefore, the second rule above is used systematically, while the first is used only when there is a choice. Although in our actual trials these simple rules have been surprisingly efficient, we recognize the need for additional research in this area. In particular, the "query cost estimator" of the type being developed by Hammer [20] would provide some additional information that could profitably be used to complement the first rule. Moreover, a better understanding of the "semantics" of both the data base and the query would probably be useful, as clearly demonstrated by Carlson [8].

Having described the purpose and function of IDA, we will now briefly present its architecture.

E. THE ARCHITECTURE OF THE IDA PROGRAM

Figure 7 presents the flow diagram of IDA. IDA is basically a loop that is executed once per relation accessed. In the next few paragraphs, we explain the main functions of each of the "black boxes" of Figure 7.

The first operation that occurs is parsing the query. This parser is extremely simple, since, as we have explained, the input language of IDA was chosen to guarantee the simplicity of the parser and to allow for an easy-to-use interface to IDA from a natural language front-end, a menu selection, a graphics package, or any other user communication front end.

Immediately after parsing, IDA decides which relations will be accessed. To do so, IDA's covering algorithm uses not only the field frames of the fields appearing in the query, but also the relation frames. The relation frames must be used since relations may need to be accessed because of indirect linkages, even though no field in the query is covered by those relations. As already explained, in choosing the relations to be accessed, the covering algorithm tries to minimize a cost function, which currently is just the number of relations to be accessed.

Once IDA has chosen which relations it will access, it enters its basic loop. First, it decides which relation it will access next; if there are none left, the complete answer has been built and is given to the user. Otherwise, the decision as to which relation to access next is taken according to the rules mentioned in the previous section. IDA examines what fields have their values specified in the user query, and tries to access first a relation where some of these fields appear. In all cases, it tries to delay as long as possible taking the maximum, or minimum, of some field values (the "* <OP> cases").

Another important heuristic involves indirect links. Let us suppose that, having applied the above rules, IDA is still left with a choice between two relations, say A and B, and that C is a relation

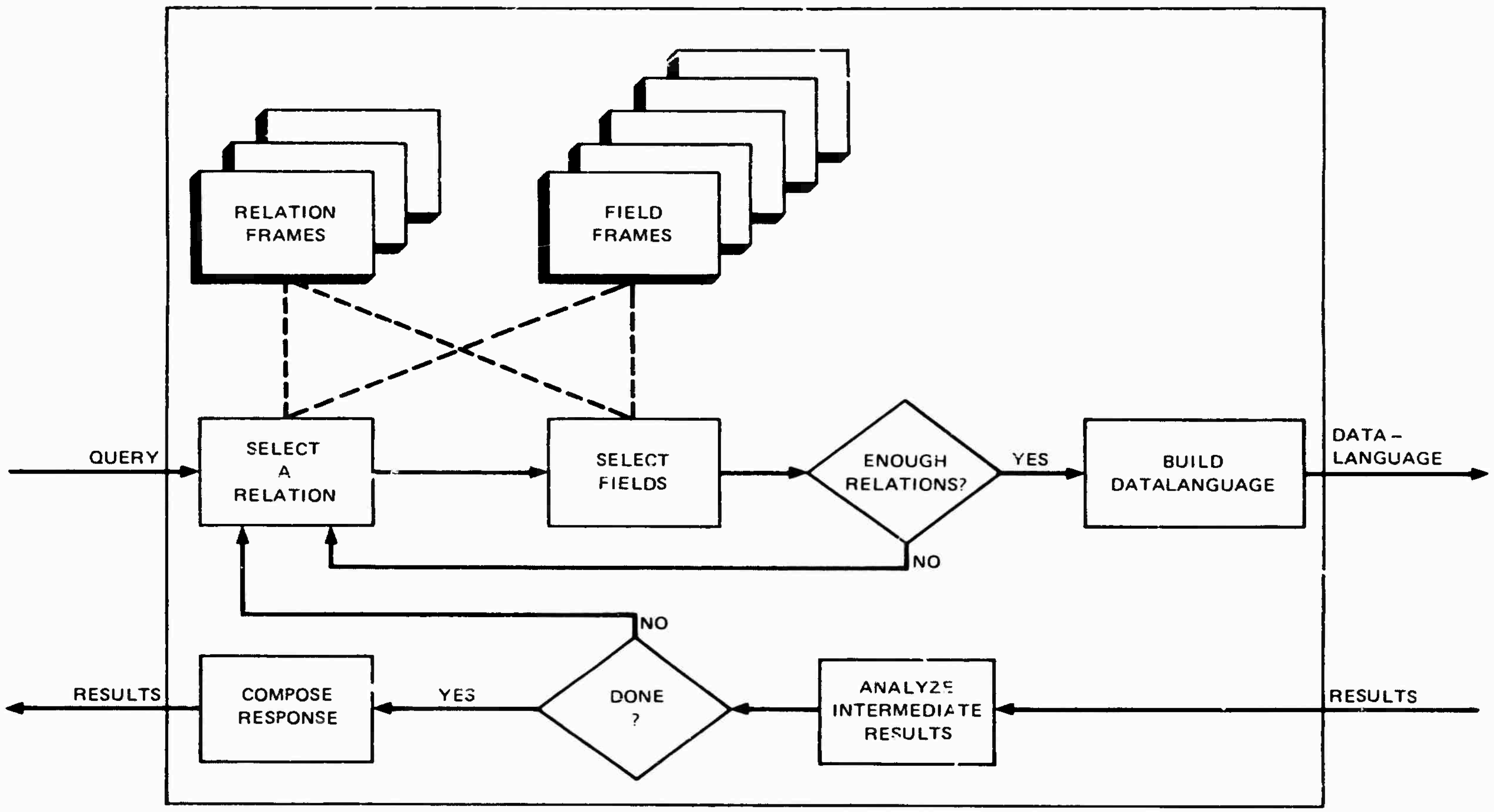


FIGURE 7 FLOW OF CONTROL OF IDA

still to be accessed. Then, IDA checks whether both A and B have a direct link with C. If one of them does not, it will be accessed first. Intuitively, if B and C have a direct link, we want to restrict the records retrieved from B as much as possible before joining them with C, and therefore we should access A first. More generally, if we have a choice between several relations, we choose to access first the one that has the most indirect links with relations not yet accessed.

Once a relation is selected, IDA decides which restrictions to send. First, it will send any boolean restriction that applies to any field in the relation. Then, it decides which values it is going to retrieve from that relation; it will ask for values of links needed for accessing later relations, of fields needed to link with already known results (from previously accessed relations), and, if they do not appear in later relations, of the fields whose values are requested by the user. IDA will send the "* <OP> field-name" query only after all other restrictions imposed by the user query have been used.

Finally, the query to the DBMS is programmed and issued. Essentially, the query portions concerning fields whose values are needed, and the boolean expressions which apply, are prepared dynamically by IDA and are incorporated into one of several prestored query templates.

When the DBMS response comes back, IDA "joins" it with the previous results and loops back to the second step. In essence, at the end of each loop, a relation has been built in IDA's local memory, which is the combination of all the information already obtained. In that sense, one may consider that IDA performs during each loop the three classical relational operations of restriction, projection, and joining. IDA performs this automatically without explicit help from the user.

Moreover, as explained in Example 5 above, IDA will also execute the "procedural attachments" during any of the steps above, as required by the structural schema.

To summarize this analysis of IDA's flow, IDA operates on a "query at a time" basis. It does not build a complete data base access program in advance, and this is one reason why IDA is reasonably time efficient. Roughly, IDA takes 100 milliseconds per relation accessed, using INTERLISP as the programming language, on a DEC KL-10 computer, under the TOPS-20 operating system.

In addition to being efficient in terms of run time, IDA is, as we have seen, quite easy to use, mainly because of the simplicity of its user interface. However, it is obvious that such efficiency has to lead to some limitations, and we explain those in the next section.

F. IDA'S LIMITATIONS AND FUTURE RESEARCH AREAS

IDA has several limitations, which we will briefly mention and analyze. The first limitation is the multi-path problem mentioned by Carlson [8] and Roussopoulos et al. [37]. A simple example of that problem arises when two relations have more than one link between them. In the simplest cases, they might have two direct links; in more complex cases, they may have several links, some or all of them being indirect. The complete solution to this problem probably requires some understanding of the semantics of the data, as explained in Carlson [8], for example. However, we feel this may not be absolutely true in all cases: sometimes it may be possible to "guess" from the query which link the user is interested in. This would occur if the structural schema indicated that some of the fields can be associated only with some of the links and not others. Then, in some cases, it could be possible to operate in essentially the same way as IDA does currently. In other cases, the query to IDA would still be ambiguous on which links to follow, and the user's intervention would still be required. For reasons of simplicity and efficiency this may be an attractive route to explore. So far, we have not pursued because we are more interested in examining how to disambiguate multi-path queries by using semantic knowledge of the data base, as suggested in Carlson [8].

Another area for future research is to extend IDA's input language to include queries not currently covered, to see whether the same basic techniques and heuristics would still apply. An example of an English question that cannot be translated into IDA query language is the following: Is there a convoy such that all ships in it have length greater than 1000 feet? The problem is that the current format for the IDA query does not admit any explicit scoping of quantifiers. It would be interesting to study such query situations, and to see whether the simple techniques used in IDA would still apply with limited modifications.

Another important research area is for IDA to access different data base management systems, which have different query languages, and even different data models. It is our contention that it would not be hard to rewrite IDA so that it would access, say, a CODASYL data base management system [10]. More challenging, and more interesting, would be to rewrite IDA so that it would access both a relational and a CODASYL data base. In other words, we would like to be able to model the data base management systems and their query languages, and use these models to build query programs in the appropriate access language. Some research has begun in this area and has been reported by Nahouraii et al. [34]. However, these authors assume the existence of the DIAM-II architecture in all the data base management systems to be used. We would like to free ourselves from such a requirement, and to assume that the user is interested in accessing data bases that are not prepared to cooperate with each other. During the coming year we will be developing such an access program for retrieval from nonhomogeneous data base management systems.

As we have mentioned, IDA operates on a heuristic step-by-step basis. In some cases, this may lead to generation of some suboptimal query programs, and may possibly even fail to produce an answer when one exists. It should be possible to use automatic program generation techniques to build a complete, optimal program of file accesses before issuing any query. Research in this area has been pursued at SRI by Furukawa [18] and has shown some promise.

Finally, we would like to point out that in all the examples we have assumed that IDA was making the joins instead of requesting the DBMS to do them. In fact, we have developed the routines to generate the DBMS queries asking for the joins to be performed. However, this creates some interesting difficulties that have to be studied further. For example, IDA should be made aware of the locations of the files, so that it will not ask for two relations to be joined that are located on different machines, which would be very expensive in time. Also, it already appears that, in some cases, the DBMS is less efficient than IDA at performing some joins; in other cases, some limit exists on the number of relations the DBMS may join as part of a single request. Consequently, a model of the DBMS's behavior will be needed to decide whether to request that the join be performed by the DBMS, or for IDA to do it itself.

G. <u>CONCLUSION</u>

We have presented the capabilities and characteristics of a data base access system that employs simple artificial intelligence techniques to free users from knowing the structure of the data base. IDA frees its users from having to know many of the peculiarities of the data base that they are using: conditional cases, redundancies, subdivisions into relations. To obtain this result, IDA decides automatically and dynamically which restrictions, projections, and joins to perform, and in what order. This may make IDA very useful in the case of large, complex data bases, where it would not be possible to build all the possible query programs in advance. In essence, IDA performs a tedious automatic programming job with reasonable simplicity and efficiency. In some cases, this approach results in a suboptimal access strategy; however, from our experience with various users, it appears that such performance is still well within acceptable limits. Additional research is needed to extend the scope of the system in the areas we mentioned; our goal is to extend it in some of the suggested areas, while still keeping its overall simplicity and efficiency.

Preceding Page Blank - NOT FILMED

V MANAGING NETWORK ACCESS TO A DISTRIBUTED DATA BASE

by Paul Morris and Daniel Sagalowicz

A. OVERVIEW

The File Access Manager is a simple system for managing access (not storage or update) of data distributed among multiple computers, each running the Datacomputer data base management system, over the ARPANET.

Because FAM has been developed to be used independently of the other components of the system, a user may interface to FAM either directly or via different front ends than those provided in LADDER.*

FAM's purpose is to provide reliable access to a distributed data base while insulating the user or calling program from inessential details concerning that data base. FAM allows the user to specify files by means of generic names: each generic file name corresponds to a primary file and perhaps also to secondary, backup copies of it. A local model of actual files and locations corresponding to these references is maintained. Given a request, FAM decides how best to meet it in terms of choosing actual files and a single location in which to assemble them. It then proceeds to make the necessary connections, logging into remote locations, opening and closing files, and moving data as required. In the event of certain classes of mishaps, FAM will take appropriate recovery action. Finally, FAM passes on the query, with such simple translations as are necessary, to the data base and returns the answer to the caller.

----

* Detailed specifications of the direct interface to FAM are provided in Appendix 3.

The basic structure of FAM is as shown in Figure 8. It is implemented as a set of LISP functions that communicates with multiple Datacomputers by using a slightly modified version of RDC, a low-level program developed by CCA that provides an interactive user with the basic functions necessary to access the Datacomputer over the ARPANET. FAM sets up an active subsidiary fork running RDC for each Datacomputer accessed. Each such instantiation of RDC receives its commands from FAM via a LISP-RDC interface and transmits them to its respective Datacomputer.

FAM maintains a local model of the distributed file system in a table stored in a disk file. This model contains information necessary for logging on and accessing files, as well as the locations corresponding to generic files. In addition, some files are identified as temporary, i.e., subject to later deletion. In addition to this model and the direct inputs, FAM's action is controlled by certain global parameters such as verbosity, which can be set by the user or calling program.

An important aspect of FAM is its ability to recover from certain types of errors. In a system involving intermachine communication in a network, certain kinds of errors inevitably occur as a result of violent termination by one or more participants. FAM deals with two basic error types. A type 1 error involves the Datacomputer crashing during a communication episode, or being down when the episode begins. The crash must be detectable at the interface. FAM responds by finding a new, hopefully more placid, location for the file in question. Access to the distributed data base system is thus protected from the vagaries of operating computer systems by redundant storing of files on different machines.

Type 2 errors are more subtle. They occur when the Datacomputer is unable to interpret FAM's command. Generall this means that the local model retained by FAM has become inaccurate. For example, a temporary file created by FAM in a previous interaction may have been deleted by another user, without a corresponding updating of FAM's local model.

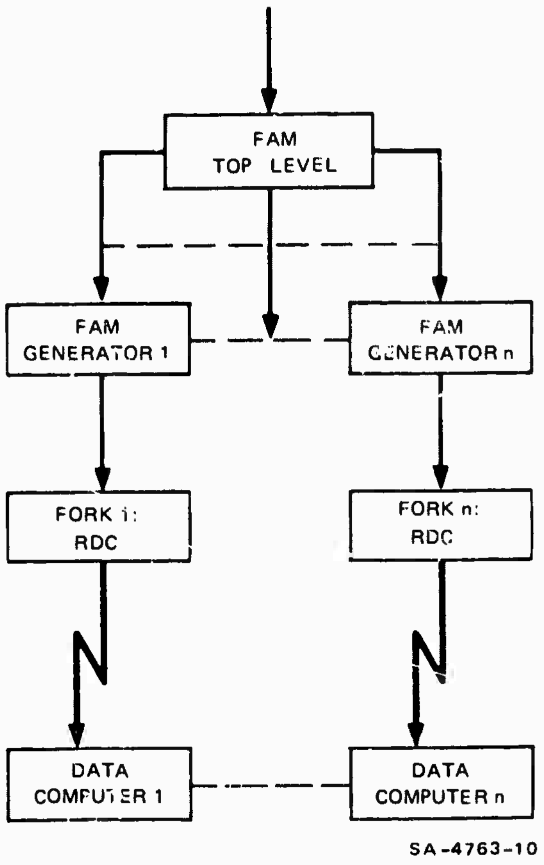


FIGURE 8 STRUCTURE OF FAM

Depending on the circumstances, FAM will take action to continue the interaction and restore the model. There is thus a tendency for inaccuracies to disappear from the model over time.

B. CAPABILITIES

This section presents a user's view of FAM's operations.* A typical user will employ the commands FAMINIT, FAMSEND, and FAMEND. (A second level of commands is described in Appendix 3; generally they allow more detailed control over FAM.)

FAMINIT is executed once to initialize the system. It may be given optional arguments specifying the model to be used (if these are not given, a default is assumed) and generic files to be immediately located and opened. FAMSEND may then be used repeatedly to query the Datacomputers and return responses. Two arguments are passed to FAMSEND: a list of generic names (of files and ports) and an expression in generic Datalanguage representing the query. Generic Datalanguage is identical to ordinary Datalanguage, except that references to files are replaced by references to generic files. The list of generic names includes all those occurring in the generic Datalanguage expression.

FAM will now search the table for the primary locations of the generic names mentioned (a location is a pair consisting of a machine and a specific file/port available on that machine).

First FAM will choose a machine in which to assemble the referenced files and ports. This is necessary because the Datalanguage expression must ultimately be handed to one machine for action. Currently, this decision is made on the basis of the least number of files and ports to be copied. Aware of copies in temporary files that it has made on previous occasions, FAM will first try to open these. In general, FAM tries to avoid doing unnecessary work. If this fails (type 2 error), FAM will recopy the file. For each location needed, FAM will attempt the following:

----

* Unless otherwise noted, statements concerning files and generic files also hold for ports and generic ports. A port is an input/output path, as well as a logical view of a file, on the Datacomputer; many Datacomputer commands do not distinguish between files and ports.

(1) If the machine is not already connected, FAM will make the network connection and log in under a suitable account.

(2) If the file or port is not open, FAM will open it (files are opened for reading, ports for writing).

(3) If steps 1 and 2 are successful, FAM is said to have accessed the location. For any generic name, if accessing of the primary location fails, the secondary locations are attempted in turn, with FAM doing the bookkeeping necessary to keep track of connected machines and opened files. Datacomputers allow a maximum number (six, currently) of opened files and ports; FAM will close least-recently used ones as necessary to open new ones.

FAM now transfers such files/ports as it needs, moving them first to the local computer -- the one on which FAM resides -- and then on to the new machine. As of this writing, the Datacomputer does not allow the transfer of files directly from one Datacomputer to another. FAM gives the copies names that reflect their origins and notes them in the model. If all goes well, the references to generic names in the generic Datalanguage are now replaced by specific names in the selected machine, and the revised Datalanguage is sent to that machine for action. The reply is read and passed back to the calling program.

When the user wants to end the session, FAMEND is called, causing the deletion of all temporary files (i.e., all files copied from one machine to another as described above), including those created during previous sessions that have not yet been deleted -- for example, in case of a computer crash in the middle of a session. One available feature allows copies to be created as permanent files which are not deleted at FAMEND. After the deletions, all the connections to Datacomputers are cleanly closed. FAM cleans up the local model and then terminates.

To summarize. FAM gives a user the semblance of dealing with a single reliable Datacomputer that is connected and whose files and ports are always open.

C. IMPLEMENTATION INFORMATION

The file access manager communicates with Datacomputers via a set of INTERLISP functions that communicates with a slightly modified version of the RDC package [17]. For each Datacomputer accessed, the subsystem RDC is started up on a separate fork, which is then synchronized with a pseudoteletype (PTY). A PTY is a local buffer that is treated by the fork as though it were a terminal; that is, commands placed in the buffer are executed by the fork, and output from the fork that would normally go to the terminal is placed in the buffer, allowing FAM to operate several RDC forks as though it were a human user with several terminals (and jobs).

To assist in the bookkeeping associated with each fork, FAM maintains separate control and data environments for each Datacomputer, implemented by the Spaghetti-Stack capability of INTERLISP [3]. A new generator is initialized whenever a Datacomputer is accessed for the first time. The generator handle is stored as the value of an atom naming the Datacomputer. Much of the information pertaining to the Datacomputer is held in the form of variable bindings within the generator. The control state of the generator is also utilized to represent certain conditions. For example, because of restrictions on the number of PTYs available, it is sometimes necessary to desynchronize a fork so that its PTY can be used elsewhere. When this happens, the generator is suspended in a state such that when it is revived it will acquire a PTY from the available pool.

To economize on stack space, when FAMINIT is executed a suitable control environment is created and a stack pointer set to it. This is intended to be close to the top level of INTERLISP, i.e., the stack is shallow at this moment. When the generators for each Datacomputer are subsequently created, they are "hung" from this point on the stack, i.e., the expressions creating them are evaluated in the control and access environment saved by the stack pointer. Individual generators can then communicate with each other through common variable bindings. The top level of FAM is able to communicate by evaluating SETQs in these access environments, as well as by passing arguments directly.

Each generator keeps a local model of the Datacomputer situation: it has a list of the open files and ports, the directories to which they belong, the directory to which the user is logged in on the particular Datacomputer, and so on.

Backtracking, needed to deal with such errors as Datacomputer crashes, is accomplished via the Spaghetti Stack: a stack pointer is set to the appropriate position for return in the event of failure. Note that during FAMEND all outstanding stack pointers created by FAM are restored and the stack is returned to its pre-FAM state.

D. THE CONTROL STRUCTURE OF FAM

The top level function of this structure is FAMSEND, which is passed a list of files and ports to be used, and a generic Datalanguage query. The locations of each file and each port are then determined from the model. Subsequently, a central machine for assembling files and ports is selected. Then, each location is handled in turn; subsequent evaluation takes place in the access environment of each machine as set up by FAMINIT. For each location, FAM decides whether it needs to do any copying and sends appropriate commands to the Datacomputers. Files are opened as needed. Finally, FAM substitutes a specific name for the corresponding generic name in the Datalanguage. All commands to each Datacomputer pass through a generator. The first time each Datacomputer is referenced, FAM establishes a copy of the generator with appropriate arguments and hangs it from the correct position on the stack. Thereafter, FAM merely revives the generator and passes on commands. Each generator evokes the functions to set up the forks, control the Datacomputer, react to errors, and perform local bookkeeping.

E. SPECIAL PROBLEMS

Many difficulties stemmed from the desirability of maintaining correctness in the local model. A single disk file was used to store the model, with open access to all users. In one instance, however, simultaneous asynchronous use allowed one FAM use to overwrite the alterations another had just made. To avoid this, when one user of FAM wishes to alter the table in which its local model is maintained, it (1) opens the file for both reading and writing in a mode that temporarily locks out other users, (2) reads the file, (3) changes the core image, (4) rewrites the file, and finally (5) closes the file so it becomes available to other FAM users. If another user tries to access the file while it is locked, this query will hang until the table becomes available.

Deleting file copies after a Datacomputer had crashed also created a difficulty. It was decided to note these not-yet-made deletions in the table and take care of them in subsequent sessions. Note that it is impossible to ensure absolute truth in the model. To see this, one need only consider the possibility of a crash of the local machine at the critical moment between making a change in the Datacomputer and updating the model. We attempted as far as possible to allow for this and to automatically make later corrections. The problems loom even larger when one contends with reconciling multiple models at different sites using the FAM system.

A minor irritation arose in connection with error messages from the Datacomputer. Although the Datacomputer supplied sufficient information in English for the user to distinguish among kinds of errors, the error code available to the program was uniform for a wide range of errors. Not wishing to parse the English messages, we were fortunately able to guess the nature of many of the errors by their contexts.

Aspects of the Spaghetti Stack forced certain inelegancies on the FAM system. Communication is awkward between the generators and FAM's top level. We suggest that generators should be able to freely access variables bound at the top level.

Another problem with generators is their loss when a control-D is executed within the generator, i.e., when the user wishes to force INTFRLISP to go back to its top level. This was prevented by redefining INTERLISP's standard GENERATE function.

F. SUGGESTED IMPROVEMENTS

One simple improvement would allow the system to choose to operate in fast or reliable modes. Currently, with each new query the system first tries to open primary locations even if secondary ones are already connected, because the information in the primary locations is presumed to be better. Since attempting access to another Datacomputer can be slow, some users might prefer to have connected secondary locations always used for certain files. A related question involves distinguishing between rapidly changing and static files, and using this information to determine the acceptability of secondary locations.

Considerable room for improvement exists in selecting the best machine for assembly of files and ports residing on multiple machines. One scheme involves modeling the "weight" of each file (i.e., the approximate time of transfer), and using it to compute a plan for a minimum-time assembly of files.

Temporary files and ports could also be dated when they are created or reused and then protected against deletion by other users until a certain period has elapsed. This would prevent the current possibility of a user's temporary files being deleted by another evocation of FAM during a job. This is actually very unlikely: the files would have to be closed at the time. In any case, FAM would recreate them as needed.

We indicated earlier that certain Datacomputer failures cannot be detected at the FAM-RDC interface. These generally involve failure of the particular job but not the overall system. When RDC times out, it is not possible to distinguish these conditions from those of a slow, heavily loaded Datacomputer. When sufficient facilities for checking on the Datacomputer are provided, FAM can be extended to recover in these situations.

The greatest deficiency in FAM is, of course, that it is a file access manager: it does not provide for the creation or update of the data base. It should also be understood that we have addressed here only a small portion of the problems arising with a true distributed data base facility.

G. CONCLUSION

We have described a file access manager that gives its users the capability of accessing files distributed over a computer network. Using a local model, FAM is able to decide which computers to use to execute the user query. It also recovers in many cases of error that may occur during the response to a query. Although FAM falls far short of the goals of a real distributed data base management system, it has shown that even a rudimentary file access program can be of considerable immediate utility.

VI A NETWORK-BASED KNOWLEDGE REPRESENTATION AND ITS NATURAL DEDUCTION SYSTEM

by Richard E. Fikes and Gary G. Hendrix

A. INTRODUCTION

Although the results of our efforts to build an efficient data access system of immediate utility have been successful as a first step, LADDER lacks many capabilities that a fully satisfactory system should have. Among many others, these include:

Language Processing Capabilities

A general ability to communicate in terms of concepts derived from those present in the data base itself. In the current LADDER system, such secondary concepts as "steaming time at maximum speed" and "steaming time at normal cruising speed" must be handled by separate, hand-coded procedures.

An ability to interpret complex elliptical expressions, anaphoric references, and definite noun phrases by appealing to knowledge about the previous dialogue and the task the user is trying to perform.

An ability to determine the appropriate format for output based on knowledge about the previous dialogue and the task the user is performing.

An ability to predict which subsections of the data base are likely to be needed soon by the user (again, based on the previous dialogue and the task being performed by the user) and to stage those subsections for more efficient retrieval.

Additional Reasoning Abilities

An ability to understand and assimilate new information given by the user, including general theorems (e.g., "All ships headed for Norfolk are 'high interest.'") and information about sources of knowledge (e.g., "The former classification of a ship is given under attribute FROM in file RECLASS on remote computer RC12.").

An ability to deal with facts and combinations of facts that were not anticipated when the system was programmed.

An ability to answer questions based on general knowledge (e.g., "Every aircraft carrier has a doctor aboard.") as well as facts stored in the data base.

An ability to answer questions about the state of the program's own knowledge and its methods of reasoning.

An ability to use complex specialist programs for reasoning processes that cannot conveniently be done by a general deduction system. For example, there might be a specialist program for generating a course for ships from one point to another in the presence of land masses.

Additional Query Capabilities

An ability to answer queries involving quantification (e.g., "Is there some American sub which is faster than every Soviet destroyer?") and scoping (e.g., "What is the largest ship in the smallest task force?").

A general ability to answer queries that have conditional representations in the data base. For example, the position of a ship that is part of a convoy might be represented only as the position of that convoy; otherwise, the position might be listed for the ship itself. Thus, the place where the ship's position can be found is conditioned on whether it is currently part of a convoy.

An ability to answer queries requiring knowledge of what the fields in the data base mean.

A general ability to answer queries involving multiple data bases of different types. In the general case, the answer to a query might involve related information stored in different data bases on different remote computers using different formats and units. To combine the data from these data bases automatically, it is necessary to model the form and meaning of the information stored in each data base so that the necessary conversions may be performed.

An ability to employ semantic transformations to allow queries to be answered in the most efficient manner possible. For example, the query "What cruisers are faster than every oiler?" could be answered by finding the speed of each cruiser, then checking the speed of a cruiser against every oiler. However, it would be more efficient to transform "faster than every" to "faster than the fastest of," find the fastest oiler, and retrieve all cruisers faster than that.

A single central problem lies in the path of developing capabilities in these areas. We need a general way of representing the meanings of things within a computer memory, and a mechanism for reasoning about them. These facilities must be complete, in the sense that the kinds of knowledge that we want to represent can be represented conveniently, and that all the logical consequences of the information represented can be determined if necessary. On the other hand, they must be efficient, both in use of core memory to represent knowledge and in use of processing time to perform deductions.

To this end, we have been developing a knowledge representation scheme based on semantic networks, called K-NET, and a problem solving system called SNIFFER (an acronym for Semantic Net Interpretation Facility Fortified with External Routines), designed to answer queries using a K-NET knowledge base.

The goal of the effort has been to create a design that allows specialized representations and deductive schemes to be used where they are effective, while providing recourse to a logically complete natural deduction mechanism when necessary. SNIFFER has been designed with the intention that most of the question answering work will be performed by special domain-dependent procedures. These specialists can take advantage of the particular topology of the K-NET structures designed to represent domain-specific types of knowledge, and thus operate much more efficiently than the general deductive mechanism. Specialist procedures also allow SNIFFER to do certain types of problem solving usually considered outside the range of conventional deduction. For example, specialists may be added that know how to extract information from conventional data bases or do scheduling and planning. In this section we seek to indicate the handles for adding specialized knowledge while concentrating on the fundamental issues of implementing natural deduction for network systems.

SNIFFER and K-NET are evolving systems. Earlier versions of them have been used as major components in larger systems developed in the SRI Artificial Intelligence Center, including the SRI Speech

Understanding System* [44]. The current versions have been designed to remedy many of the inadequacies of the previous systems, and will be incorporated into a system for sophisticated data access during the coming year.

To help the reader relate our work to other knowledge representation facilities and problem solving systems, we begin by presenting the distinguishing and characterizing features of our system before focusing on a more detailed overview that elaborates on these features and provides concrete examples.

B. CHARACTERIZING FEATURES OF K-NET

K-NET provides facilities for creating a partitioned semantic network of labeled nodes connected by labeled unidirectional arcs. A node represents an entity in the world being modeled and an arc represents a binary relationship between the nodes that it connects. For example, the nodes John** and Men in Figure 9 represent a man John and the set of all men, respectively. The arc labeled "e" from John to Men indicates that John is an element of the set of men. Relationships can be considered to be entities and be represented by nodes with "case" arcs pointing to the participants in the relationship. For example, node Q represents the ownership relationship (situation) existing between John and the automobile "Ole-Black" over the interval from time t1 to time t2. K-NET can be characterized by the following list of features:

- Facilities are provided for representing multiple "worlds" and the relationships among them. In particular, the network can be partitioned into subnets (called spaces). Spaces can be hierarchically embedded by treating an entire space at one level in the hierarchy as a single node in a space at the next higher level. A "context" mechanism exists that allows only a given set of spaces to be "visible" to the retrieval procedures at any one time.

---

* This research was sponsored by DARPA under contract DAAG29-76-C-0011 with the U.S. Army Research Office.

** The names of nodes and arcs are underlined to distinguish them from the concepts they encode.

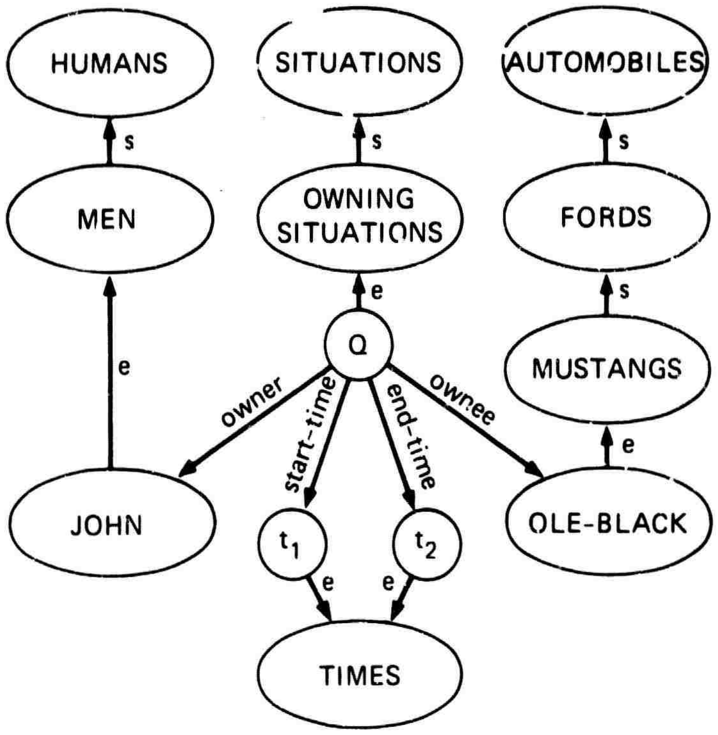


FIGURE 9 A SIMPLE SEMANTIC NETWORK

Examples of alternative worlds include those contained in a disjunction, or the world composed of the set of beliefs that John has about Sally as opposed to the world composed of the set of beliefs that Sally has about herself.

* The expressive facilities of the representation scheme include those of the first order predicate calculus, including existential and universal quantification. (Higher order predicates are also representable in K-NET, but only trivial interpretation facilities exist for them in SNIFFER.) That is, the knowledge base can contain statements represented as negations ("John does not love Mary."), disjunctions ("John loves either Sally or Sue."), or implications ("If Sue answers John's phone call, then John will ask Sue for a date."), and containing arbitrary nestings of existential and universal quantifiers ("Every boy has been in love sometime.").

* Taxonomies of sets are modeled by the topology of the network so that they become the basic skeleton upon which the knowledge base is built. For example, one can directly represent the relationships "Ford is an element of Companies distinct from G.M." and "Mustangs is a subset of Automobiles distinct from Model-T's." One can also associate with a set characteristic properties common to all elements of the set, such as "All Mustangs are built by Ford."

* Procedures may be attached to the network to interface it to other knowledge sources such as conventional data bases or arithmetic algorithms. When called by SNIFFER, these procedures extend the network by creating new nodes and arcs representing the information acquired from the other sources. Links to these procedures are explicitly represented in the network so that their existence and role can be reasoned about and discussed by the system.

* The network provides indices that facilitate associative retrieval of the relationships in which any given knowledge base entity is involved. For example, retrieval of all females that John loves can be indexed through the node representing John, the node representing the set of loving relationships, or the node representing the set of females. The basic mechanism is one that allows immediate access to all of a node's incoming and outgoing arcs that are visible in any given set of spaces.

C. CHARACTERIZING FEATURES OF SNIFFER

SNIFFER is a "natural" deduction system (as in Bledsoe [2]) that is given two net structures as input, one representing a knowledge base and the other representing a query (usually a translation of a question originally stated in English). It treats the query as a pattern and attempts to find instances of the pattern in the knowledge base, or equivalently, it treats the query as a theorem to be proved and attempts to find instantiations for its existentially quantified variables. Results are returned in the form of sets of "bindings" to the variables in the pattern. For example, the question "Who does John love?" is translated into a net structure representing the pattern "John loves x" (or the theorem (Ex)[Loves(John,x)]), and SNIFFER returns bindings for x such as (x, Mary). Answers may either be retrieved from the knowledge base or derived using knowledge base theorems and procedures. SNIFFER can be characterized by the following list of features:

* Associative retrieval of relationships from the knowledge base is performed using the K-NET indexing facilities.
* Efficient, special-purpose deductive procedures are used for extracting information from the K-NET taxonomies. For example, if the knowledge base indicates that x is an element of the set of Mustangs, that Mustangs are a subset of the set of sports cars, and that sports cars are a subset of the set of automobiles, then SNIFFER can conclude that x is an automobile by using procedures that follow the chain of elementOf and subsetOf arcs, thereby bypassing the more cumbersome, general-purpose deductive machinery.
* Facilities are included for answering questions and using knowledge base statements composed of conjunctions, disjunctions, and implications, containing arbitrarily embedded universally and existentially quantified variables.
* Queries and knowledge base statements are processed in the "natural" form in which they are input, without converting into a canonical form such as clause form or prenex normal form. This capability eliminates "explosive" conversions (such as converting the disjunction (a&b&c) v (d&e&f) v (g&h&i) into clause form which consists of 27 clauses, each containing 3 disjuncts) and unnecessary conversions (such as conversion of a disjunctive question's complex disjuncts when one of its simple disjuncts can easily be shown to be true). In addition, the

intuitiveness and heuristic value of the form in which statements are input (as implications, for example) is maintained.

* A logically complete set of natural deduction rules are used that reason backwards from the question. These rules use such techniques as case analysis, hypothetical reasoning, and the establishing of subgoals. For example, to answer a question that is in the form of an implication, SNIFFER might use hypothetical reasoning by assuming the implication's antecedent and then pursuing a proof of the consequent as a subgoal.

* A flexible coroutine-based control structure allows the construction of alternative proofs in a pseudo-parallel manner, with results being shared among the alternatives. Each partial proof has its own local scheduler to determine how its proof attempt should be continued. There is an executive scheduler that uses information supplied by the local schedulers to determine which partial proof is to be given control at each step. The various schedules provide the facilities necessary to allow reasonable heuristic guidance of the total deduction and retrieval process.

* User-supplied procedures may participate in the attempt to find answers in two ways. First, procedures included in the K-NET knowledge base may be invoked to access information in knowledge sources that are external to K-NET. Second, SNIFFER allows the inclusion of user-supplied procedures that extend the system's problem solving capabilities. Such procedures may add heuristics to the deductive strategies or even integrate new knowledge sources into the system, such as data bases and planners. Facilities are available to these procedures for creating alternative proofs, manipulating schedules, altering priorities, and establishing "demons" so that the user can create strategies that augment and interact with those that already exist in the system.

* SNIFFER is implemented as a "generator" (see Teitelman[42]) so that after returning an answer it can be restarted to seek a second answer to a query. For example, given the question "Who owns a Mustang?" SNIFFER may first produce the answer "John", then be "pulsed" again to produce "Mary", etc. This style of answer production allows the user to examine each answer as it is produced and dynamically determine whether additional answers are needed.

* "No" answers are determined by finding an affirmative answer to the question's negation. For example, if given the question "Does John love Mary?", SNIFFER will attempt to prove "John does not love Mary" in addition to attempting to prove "John loves Mary."

D. <u>OVERVIEW DESCRIPTION OF K-NET</u>

In this section we will describe and illustrate how K-NET is used to encode knowledge. Throughout the section reference will be made to the example knowledge base shown in Figure 10, which represents some facts about automobiles.

1. <u>Taxonomies</u>

Major portions of the semantics of a task domain can often be expressed naturally by a taxonomy of sets that indicates the major sets of objects in the domain and the relationships between the sets. The power of the taxonomy can be enhanced further by the inclusion of statements that specify necessary and/or sufficient conditions for membership in the sets. K-NET provides the following facilities designed specifically for encoding such taxonomies. <u>S</u> arcs indicate "subset of" relationships. For example, the <u>s</u> arc in Figure 9 from the <u>Men</u> node to the <u>Humans</u> node indicates that the set of men is a subset of the set of all humans.

Most sibling subsets described in taxonomies are disjoint. Arcs labeled "ds" are used in K-NET to represent this disjointness property in a concise and easily interpretable manner. A <u>ds</u> arc from a node <u>x</u> to a node <u>z</u> indicates that the set represented by <u>x</u> is a subset of the set represented by <u>z</u> and that the <u>x</u> set is disjoint from any other set represented by a node with an outgoing <u>ds</u> arc to <u>z.</u> For example, the <u>ds</u> arcs in Figure 10 emanating from the <u>Humans</u> and <u>Companies</u> nodes indicate that the set of humans and the set of companies are disjoint subsets of the set of legal persons.

Since each node in most taxonomies represents a distinct entity, and in general an entity can be represented by any number of nodes in a K-NET, arcs labeled "de" (for "distinct element") are used to indicate that two or more nodes each represent a distinct element of a set. In particular, a <u>de</u> arc from a node <u>x</u> to a node <u>z</u> indicates that the entity represented by <u>x</u> is an element of the set represented by <u>z</u> and that the <u>x</u> entity is distinct from any other entity represented by a node that

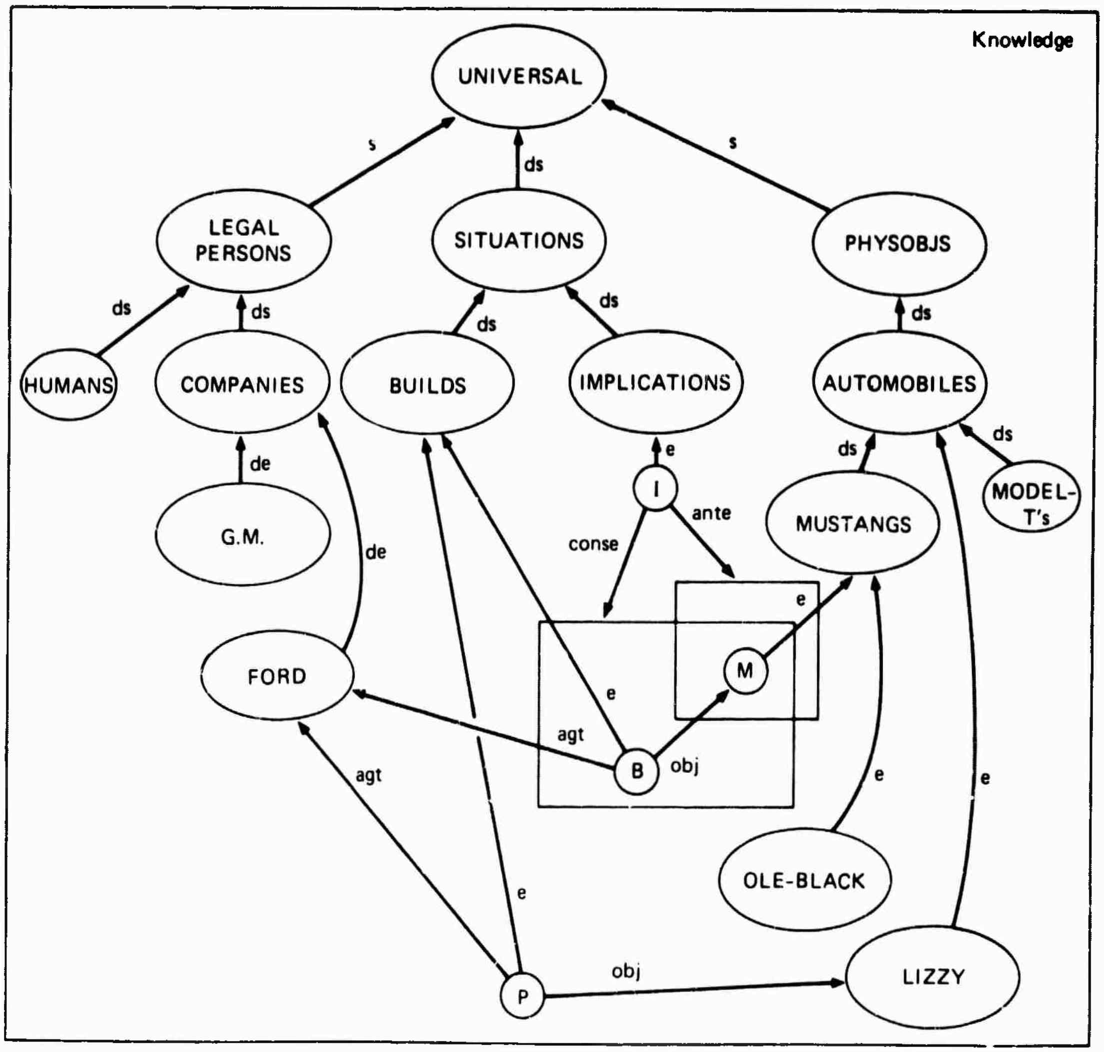


FIGURE 10 AN EXAMPLE KNOWLEDGE BASE

has an outgoing de arc to z. For example, the de arcs in the figure emanating from the G.M. and Ford nodes indicate that G.M. and Ford are distinct members of the set of companies.

E arcs are used to indicate "element of" relationships without making a commitment to distinctness. For example, Fred, Jill, and Mary may be known to be distinct elements of RIDERS, the set of people that rode to the airport in Fred's car. If some fact is known about the driver of the car and the identity of the driver has not yet been determined, then a node D representing the driver may be linked to node RIDERS by an e arc. The node D can be used to encode information about the unnamed driver without specifically indicating which of the distinct elements of RIDERS is the driver.

2. Situations

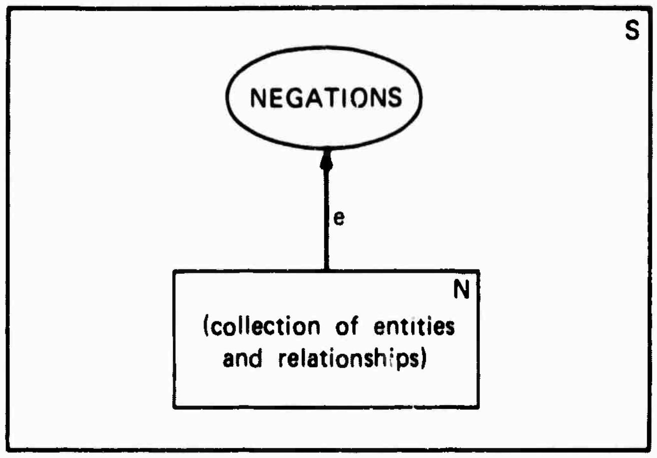
SNIFFER assumes that relationships other than "element of" and "subset of" are represented by nodes having outgoing case arcs pointing to the participants in the relationship (such as node P in Figure 10, which represents the relationship "Ford built Lizzy"). This representational convention allows an arbitrary amount of information to be stored with a relationship (using outgoing case arcs) and allows associative retrieval of the relationship using the network's indexing facilities. Such relationships are grouped by type into sets and these sets are considered to be subsets of the set of all "situations." For example, BUILDS (the set of all situations in which building takes place) and IMPLICATIONS are disjoint subsets of SITUATIONS in Figure 10, and node P represents an element of the BUILDS set, a particular building situation in which Ford is the agent and Lizzy is the object built. (The situation represented by P took place over an interval of time from StartTime to EndTime. These time cases would be present in a more complete description of P.)

3. Spaces and Vistas

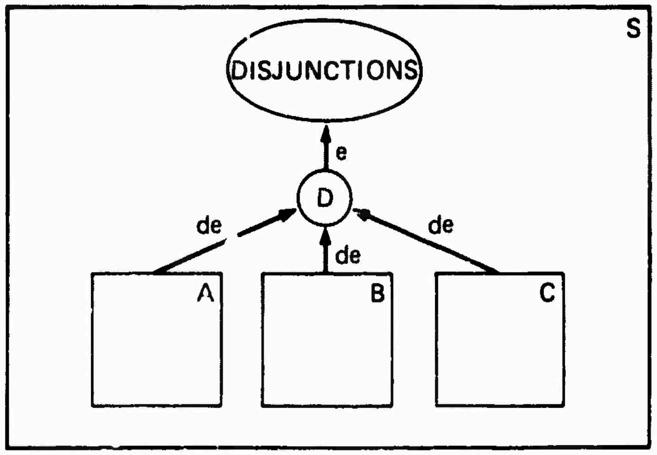
Perhaps the primary feature that distinguishes K-NET from other semantic networks is that a net can be partitioned into subnets, and relationships among the subnets can be explicitly and easily represented (see [24] for an introduction to this concept). All nodes and arcs in a K-NET are "elements" of at least one "space" (i.e., subnet). In the figures, such spaces are depicted by boxes. For example, node P in Figure 10 and the obj arc from P to Lizzy are elements of the Knowledge space. A space can be (and usually is) a node in some other space. For example, in Figure 10 the conse arc from node I points to a node in the Knowledge space that is itself a space. When retrieving information from a network, it is convenient to have only a specified list of spaces, called a "vista," visible to the retriever. For example, the vista that would typically be used when retrieving information from the space pointed to by the conse arc in the figure consists of the space itself and the Knowledge space.

4. Negations, Disjunctions, and Implications

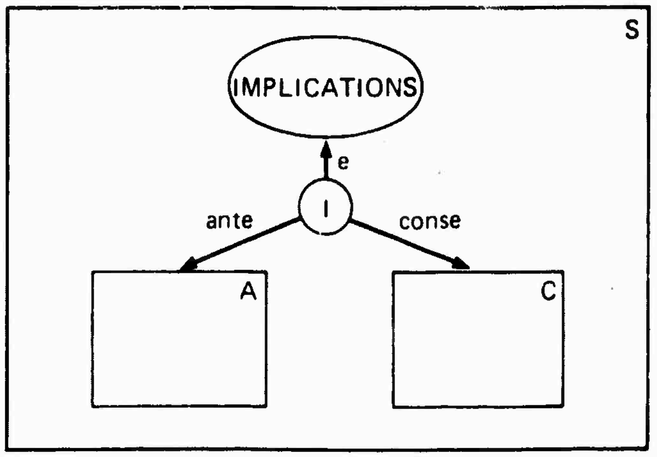
A representation scheme for negations, disjunctions, and implications must allow one or more "worlds" to be described and a relationship to be asserted among the worlds (e.g., that at least one of them is true). K-NET's partitioning facilities provide the required capabilities for creating just such a scheme. ELSE !H! _ FULL; A negation occurring in some space s describes a collection of entities and relationships, and asserts that no collection satisfying the description can exist in the world represented by space s. We represent such a negation as shown in Figure 11a by creating a space to describe the collection, and by adding the created space to space s as a node with an outgoing e arc to negations, the node that represents the set of all negation relationships. For example, the statement "G.M. does not build convertibles" would be represented using a space describing a collection consisting of an entity C, an elementOf relationship between C and the set of convertibles, and a build relationship with agent G.M. and object C.



a. NEGATIONS (¬N)



b. DISJUNCTION (A ∨ B ∨ C)



c. IMPLICATION (A → C)

FIGURE 11 ABSTRACTION OF LOGICAL CONNECTIVES

A disjunction occurring in a space s describes alternative collections of entities and relationships, and asserts that entities and relationships satisfying at least one of those descriptions exist in the world represented by space s. As shown in Figure 11b, we describe each disjunct in a separate space and represent a disjunction as a set of such disjunct spaces.

An implication occurring in a space s describes two collections of entities and relationships, and asserts that if entities and relationships exist in the world represented by space s that satisfy the first of the two descriptions (the antecedent), then entities and relationships satisfying the second description (the consequent) also exist in that world. We represent an implication as shown in Figure 11c by a node with outgoing case arcs to spaces containing the descriptions of the antecedent and consequent. More concrete examples of implications will be presented in the next subsection.

5. Quantification

One of the important features of K-NET is that it provides facilities for representing arbitrarily nested existential and universal quantifiers. Existential quantification is a "built-in" concept in the sense that we take the occurrence of an element (i.e., a node or arc) in a space to be an assertion of the existence with respect to that space of the entity or relationship represented by the element. In particular, if an element occurs in the system's "knowledge space," then that element represents the system's belief that a corresponding entity or relationship exists in the domain being modeled.

Existential quantification and negation could be used to represent any universally quantified formula* (Ax X)P(x) by making use of the following transformation:

$$(Ax \epsilon X)P(x) \;=\; \sim\sim[(Ax \epsilon X)P(x)] \;=\; \sim[(Ex \epsilon X)\sim P(x)] .$$

The K-NET representation of the transformed formula is shown in Figure 12.

----

* Note: we write "A" for universal quantification, "E" for existantial quantification, and "=" for equivalence.

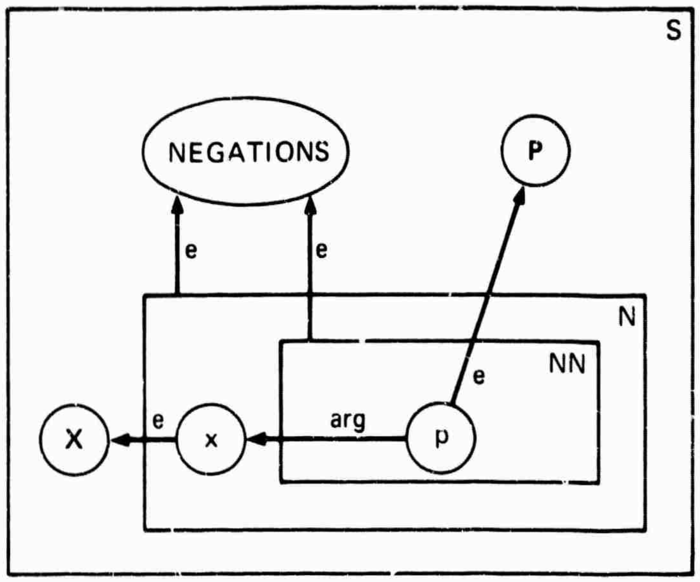


FIGURE 12 $\sim[(\exists x \in X)\sim P(x)]$

Although this representation is logically sound, it is extremely unappealing intuitively. The following transformation suggests a more attractive representation:

$$(Ax \in X)P(x) \ = \ (Ax)[(x \in X) \Rightarrow P(x)] \ .$$

That is, any universally quantified formula can be represented as an implication whose antecedent specifies the "typing" of the universally quantified variable and whose consequent specifies the statement that is being made about any entity that satisfies the type restrictions.

A distinguishing feature of the universally quantified variable x in this implication is that it occurs in both the antecedent and the consequent. We have made use of this feature by adopting the convention in K-NET that if a node occurs in both the antecedent and the consequent spaces of an implication, then we consider it to be the representation of a universally quantified variable. This convention is, in fact, used as the primary means of representing universal quantification in our system.

When the main connective of a formula is an implication, it is not necessary to embed the formula in another implication to represent the universal quantification. That is:

$$(Ax \epsilon X)[Q(x) \Rightarrow R(x)] =$$
$$(Ax)\{(x \epsilon X) \Rightarrow [Q(x) \Rightarrow R(x)]\} =$$
$$(Ax)\{[(x \epsilon X) \& Q(x)] \Rightarrow R(x)\}.$$

Figure 10 shows the K-NET representation of a concrete example of such an implication, namely the statement "For all M in the set of Mustangs, there exists a B such that B is an element of the set of Builds situations, the agent of B is Ford, and the object built is M."

Arbitrary nesting of quantifiers may be achieved by placing implications in the consequent spaces of other implications. For example:

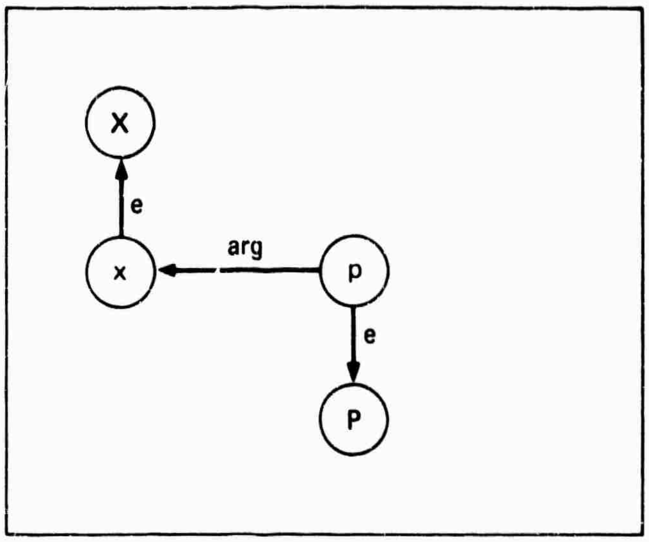
$$(Ax \epsilon X)(Ey \epsilon Y)(Ax \epsilon Z)P(x,y,z) =$$
$$(Ax)\{x \epsilon X \Rightarrow (Ey)[y \epsilon Y \& (Az)(z \epsilon Z \Rightarrow P(x,y,z))]\} .$$

Figure 13 summarizes the conventions for representing quantification by contrasting the K-NET representations of $(Ex \epsilon X)P(x)$ and $(Ax \epsilon X)P(x)$.

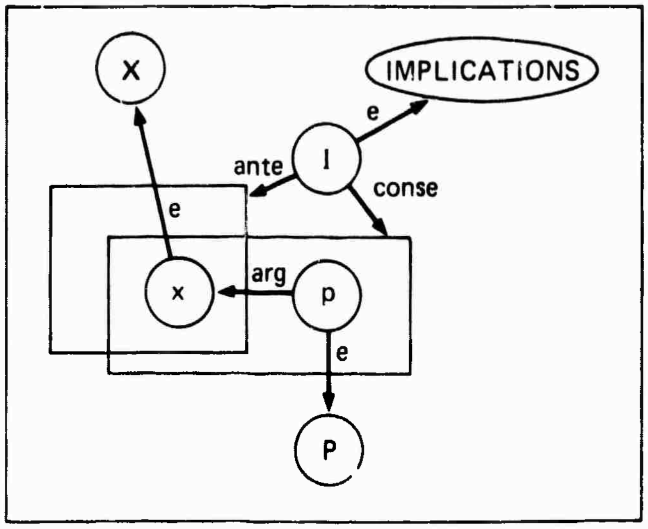
6. Procedural Augmentation

For many applications, including, in particular, data base access, it is important for the system's knowledge base to include sources of information such as relational data bases or arithmetic algorithms external to the K-NET nets. (See Reiter [36] for another example of an inference system designed to interact with a relational data base.) We have adopted a set of conventions in K-NET for describing links to such external knowledge sources.

The links to external knowledge sources are represented by "theorems" (i.e., implications containing universally quantified variables) in the system's knowledge space that have the form exemplified by the network shown in Figure 14. Such theorems are interpreted to mean that if there is a successful application of the indicated function to a set of arguments that satisfy the description given in the antecedent, then the arguments and the results returned by



$(\exists x \epsilon X)P(x)$ or
$\exists x[x \epsilon X \wedge P(x)]$



$(\forall x \epsilon X)P(x)$ or
$\forall x[x \epsilon X \rightarrow P(x)]$

FIGURE 13 EXISTENTIAL AND UNIVERSAL QUANTIFICATION

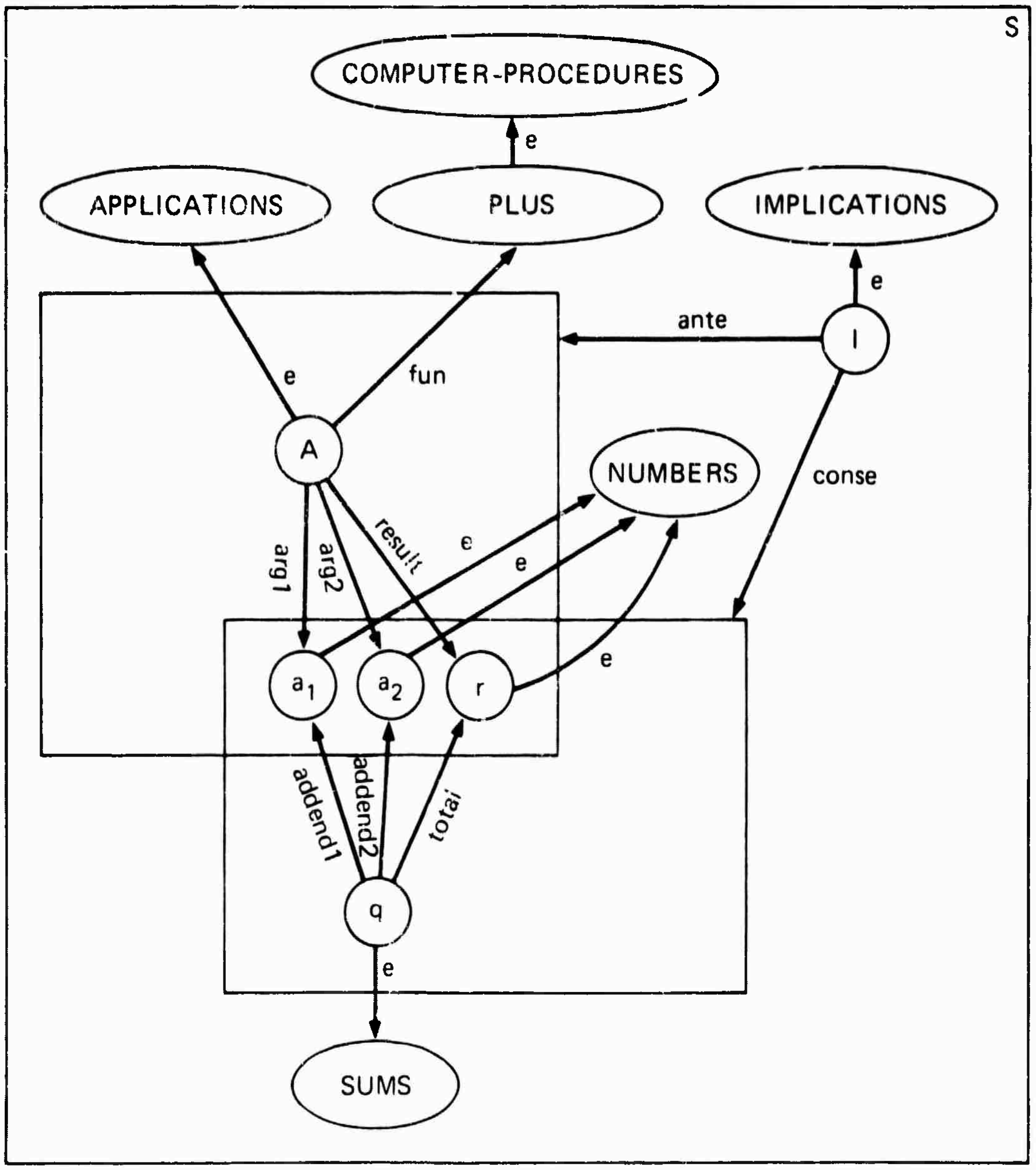


FIGURE 14 LINKING RELATION SUMS TO PROCEDURE PLUS

the function can be used to create relationships and entities satisfying the description given in the consequent.

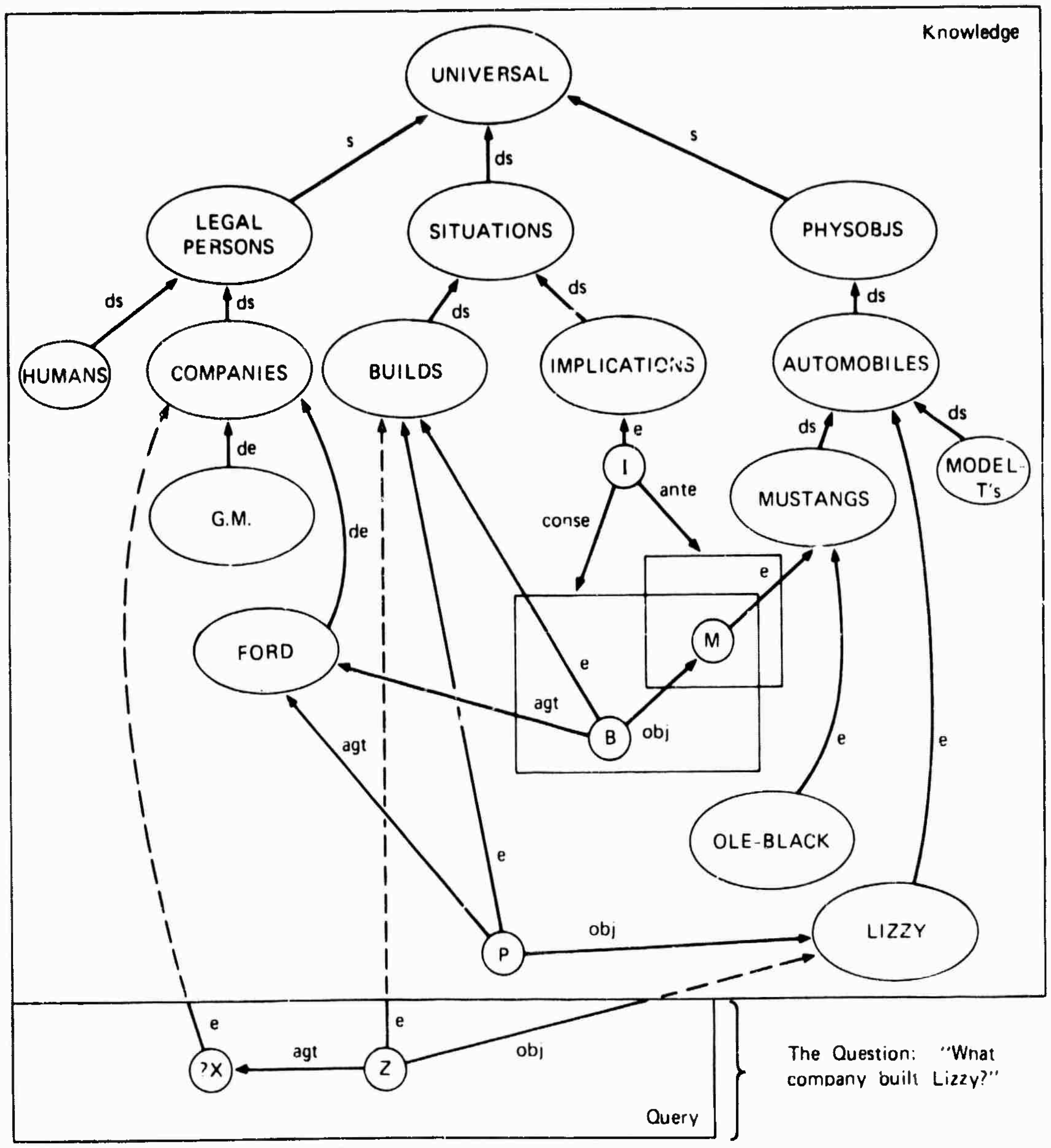
The particular theorem of Figure 14 indicates that an application of INTERLISP's PLUS function can be used to produce new instances of the SUMS relation in the net. This theorem makes it unnecessary for all the instances of the SUMS relation to be explicitly represented in the knowledge base. When SNIFFER attempts to match a pattern involving the sum of two numbers, it can use this theorem to form a call of the PLUS function and to translate the results of that call into the desired SUMS relationship. The manner in which SNIFFER uses knowledge about the Applications set to create new relationships from the results of procedure calls is discussed below in the section on special-purpose binder tasks.

E. OVERVIEW DESCRIPTION OF SNIFFER

This section describes and illustrates the basic features used by SNIFFER in retrieving and deriving information from K-NET structures. We begin by considering how SNIFFER is invoked and by illustrating how it would go about solving two simple problems. Attention is then turned to the overall control structure and to the operations performed by various components.

1. Introduction

SNIFFER is given as input a vista representing a query (the QVISTA) and a vista representing the beliefs that are to be considered true while answering the query (the KVISTA). Like other vistas, the QVISTA and KVISTA are lists of spaces. In aggregate, the nodes and arcs of the various spaces in the QVISTA describe a set of entities (i.e., objects and relationships) whose existence is to be established in the KVISTA. If a set of such entities can be found to exist, then SNIFFER returns a list of "bindings" that link the QVISTA descriptions to their KVISTA instantiations. Otherwise, SNIFFER attempts to prove that no such collection of entities can exist, so that a negative response can be given.



KVISTA = (Knowledge)
QVISTA = (Query)

FIGURE 15 WHAT COMPANY BUILT LIZZY?

For example, Figure 15 shows a KVISTA and a QVISTA for the query "What company built Lizzy?" Given this QVISTA, SNIFFER seeks an element of the Builds situations set having both Lizzy as its object and an element of the Companies set as its agent. The Builds situation represented by node P in the KVISTA is found by using the incoming e arcs to the Builds node as an index, and a "Yes" answer is generated with P as the binding for node Z and the Ford node as the binding for node ?X. The "Yes" answer indicates that the question was based on a true premise and the binding for ?X is the actual value that was sought.

Given the KVISTA and QVISTA shown in Figure 16, SNIFFER must carry out a derivation to answer the query using the KVISTA theorem "All Mustangs were built by Ford." The theorem is found by indexing on the incoming e arcs to the Builds node. A unification process determines that the relevant instance of the theorem is one in which the universally quantified variable M is replaced by Ole-Black. The theorem allows a new Builds situation to be asserted if it can be shown that Ole-Black is an element of the Mustangs set. A subproblem is created to find that ElementOf relationship, and when the subproblem is solved, the new Builds situation is asserted and the desired bindings are assigned. In particular, node ?X is again bound to Ford and Z is bound to the node representing the newly derived Builds situation.

2. Control Structure

As an introduction to SNIFFER's control structure, consider the following simplified description of how the system goes about answering queries. The basic process consists of selecting an unbound QVISTA arc and finding a match for the selected arc in the KVISTA. The matching arc then implies matches for the nodes at each end of the selected QVISTA arc. After each arc is bound, the process is repeated until all the arcs and nodes of the QVISTA have been bound.

This conceptually simple process is complicated by a number of factors. At each step in the process there are typically many

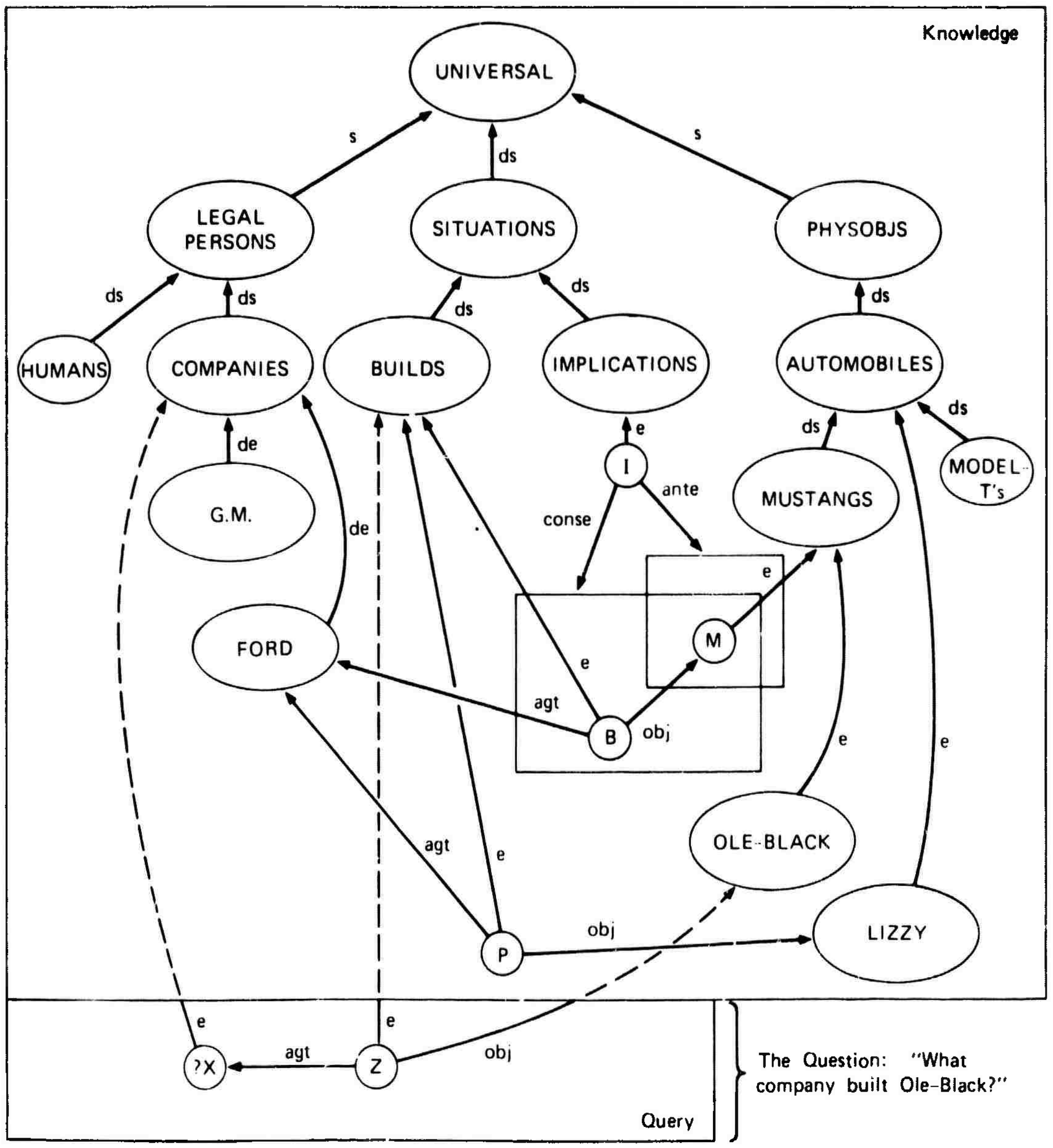


FIGURE 16 WHAT COMPANY BUILT OLE-BLACK?

alternatives that may be followed. For example, any of the unbound arcs in the QVISTA might be selected for consideration and each of these might be successfully bound to many KVISTA arcs. Another complicating factor is that some structures in the QVISTA will have no matches in the KVISTA, even though their existence is implied by statements in the KVISTA. Deductive machinery must be invoked to derive explicit representations of these implied structures. Within the deductive machinery, choices must be made between alternative strategies for pursuing a derivation and among the collection of KVISTA statements that could possibly be used to derive the desired matching structure.

The control structure that we have evolved for SNIFFER allows these various alternatives to be pursued in a pseudo-parallel "best first" manner. K-NET's partitioning facilities and INTERLISP's coroutines are used to create a system environment that allows each alternative to have its own subproblems, assumptions, and derived results, and for the choices among these alternatives to be guided by both built-in and user-supplied evaluation functions.

3. The Environment Tree and Task Agendas

SNIFFER proceeds by building a tree of alternative proofs, each node of which represents a data environment that includes a set of choices of bindings for QVISTA elements and derivation strategies. Each time a choice is to be made in an environment, an offspring environment is created and the results of the choice are established in the offspring. For example, if a binding for a QVISTA element is found, then an offspring environment will be created in which the binding will be assigned. The search for additional bindings can then be continued in the parent environment, but SNIFFER is committed to the assigned binding in the offspring.

Included in each environment is a task agenda (patterned after the agenda mechanism in Bobrow and Winograd's KRL-0 [5]) that defines a given number of priority levels and allows a list of tasks to be stored at each priority level. The SNIFFER executive typically proceeds by selecting an environment to which to give control, and then running the highest-priority task on the selected environment's agenda. Each task is composed of a LISP function and a set of arguments on which the function is operating. Typical tasks look for KVISTA descriptions matching a given QVISTA description or, if necessary, initiate derivations to deduce new explicit descriptions from implicit descriptions contained in KVISTA "theorems."

The executive also has its own task agenda that is used to determine what to do at each step. Initially, this agenda has three tasks on it: one to initialize an environment tree to seek "Yes" answers to the query, one to initialize an environment tree to seek "No" answers to the query, and the one mentioned above that selects an environment, runs the task defined by that environment's agenda, and reschedules itself.

The agenda associated with the top environment in an environment tree initially contains a single task that selects for consideration unbound arcs that lie in the QVISTA. Each time the selector task is restarted, it selects another QVISTA arc, creates a "binder" task that will seek bindings for the selected arc, schedules the created task, and reschedules itself.

When a binder task finds a KVISTA arc that is a "candidate" (i.e., potential) binding, it creates a new offspring environment in the environment tree that is a copy of the parent, assigns the binding in the offspring environment, and reschedules itself in the parent environment. Hence, at any given step, each terminal environment in the tree includes a partially formed alternative answer to the query.

Provisions have been made for attaching "demon" functions to QVISTA nodes and spaces in an environment. Demons attached to a QVISTA node, which are "fired" when a binding is assigned to the node, allow binder tasks to "pause" until other bindings have been assigned that can be used as indices. Demons attached to a QVISTA space, which are fired when bindings have been assigned to all the arcs and nodes in that space, are useful in completing derivations and returning results. For example, demons are attached to each QVISTA space in the initial environment of an environment tree. When the last of these demons fires in an environment, bindings will have been assigned to all QVISTA elements in that environment and an answer can be generated. The last demon causes the answer to be generated by scheduling an appropriate task on the executive's agenda.

When an offspring environment is created, it inherits copies of its parent environment's data structures, including the agenda, demons, and list of assigned bindings. If a task or demon represents a "paused" coroutine that will be "resumed" when the task is run, then copying it conceptually produces a copy of the coroutine so that the original task or demon and the copy can run independently in their respective environments. For example, if a binder task is in a state such that it will consider relationship R as the next candidate binding and it is copied into an offspring environment's agenda, then the copy will also independently consider R as the next candidate binding. Similarly, a demon can be independently fired in each environment in which bindings for all the space's elements have been assigned. This powerful capability is implemented using the "spaghetti stack" facilities found in INTERLISP [3].

4. Binder Tasks and User-Supplied Specialists

The Selector task in each environment's agenda selects unbound QVISTA arcs and creates binder tasks that seek bindings for the selected arcs. The procedures used in the binder tasks embody the system's retrieval and derivational mechanisms.

a. Domain-Specific Augmentation

The primary way in which SNIFFER can be augmented and adapted to a particular problem domain is by providing additional procedures that can act as "expert" binder tasks for specialized classes of relationships. Such experts may add heuristic guidance to the deduction process or add completely new sources of knowledge.

For example, a binder task for ownership relationships might add heuristic guidance by knowing that objects usually have a unique owner. This task would look for bindings by following indices from the object to its owner rather than from the person to all the objects he/she owns or from the set of all ownership relationships.

Another expert binder task might be written for the relationship between a person and his telephone number. Rather than look for the person/number relationship in the K-NET, this procedure might look it up externally in a phone book file. The procedure would then create new structures in the KVISTA to encode the retrieved information and use this new structure in the binding.

b. Strategy Selectors

When a QVISTA arc has been selected, it is passed through a set of "strategy selectors," each of which is a function that can create a binder task for the arc and indicate whether additional selectors should be consulted. When a new function for finding bindings is added to the system, a strategy selector is written for it and added to the set of selectors. These strategy selectors provide a generalized form of pattern directed invocation of the binder tasks.

When no "specialist" binder task is available for a selected arc, a general-purpose binder task is created that can seek bindings for any relationship or its negation using natural deduction theorem proving strategies. It uses the net's indexing facilities to first find all atomic statements (i.e., relationships other than disjunctions, implications, or negated conjunctions) that contain possible bindings for the selected arc and then all nonatomic statements that can be used to derive bindings for the selected arc. For example, the general-purpose binder task for arc Z--e-->Builds in Figure 16 would consider incoming e and de arcs to the Builds node as candidate bindings.

c. Ramification

When a binder task finds a candidate binding, it can apply the following "ramification" rules to determine what other bindings are directly implied by the candidate. First, if two arcs are to be bound to each other, then the from-node of the first arc must be bound to the from-node of the second arc, and the to-node of the first arc must be bound to the to-node of the second arc. Second, we assume that a node can have at most one outgoing case (i.e., nontaxonomic) arc with any given arc label. Therefore, if two nodes are to be bound to each other and both nodes have outgoing case arcs with a common label, then those case arcs must also be bound to each other. For example, if in Figure 15 arc P--e-->Builds were the candidate binding for arc Z--e-->Builds, then bindings would be implied for nodes Z and ?X, and for the agt and obj arcs.

If a candidate binding implies a binding that is inconsistent with an existing binding (for example, one that assigns two different bindings to some QVISTA node, where ds and de arcs in the taxonomies indicate that the two bindings represent distinct entities), then the candidate can be rejected and another one sought. Hence, this ramification process acts both as a powerful and efficient filter for candidate bindings and as a producer of new bindings.

d. Self-Scheduling

The decision as to which binder task should be given control in any given environment is made by allowing each such task to determine the priority level at which it is scheduled on the environment's task agenda. A task makes this determination by assessing the difficulty of finding bindings for its QVISTA arc based on estimates of the number of indices (i.e., matching arcs) available in the KVISTA, knowledge about the semantics of the relationship being sought, knowledge about the effectiveness of the task's search method, etc. User-supplied specialists may be written that are particularly adept at such assessments. The basic goal of the overall strategy is for the system to first seek bindings for those QVISTA arcs that are most highly constrained.

5. Deriving Bindings for ElementOf and SubsetOf Relationships

Included in SNIFFER is a set of functions embodying the semantics of the taxonomic relationships represented by e, de, s, and ds. These functions provide the following eight services:

> Given a node representing some entity x, they can generate nodes representing entities y such that x is an element of y, y is an element of x, x is a subset of y, or y is a subset of x.
>
> Given two nodes representing entities x and y, they can determine whether x is an element of y, y is an element of x, x is a subset of y, or y is a subset of x. Possible answers are "Yes", "No", and "Unknown".

The algorithms used follow chains of s and ds arcs applying recursive rules such as the following:

> Two sets are disjoint if each of the nodes representing them has an outgoing ds arc to the same node, or if the sets are each subsets of disjoint sets.

These functions are used in SNIFFER wherever information is needed about SubsetOf or ElementOf relationships. In particular, they are used by the general-purpose binder task to find candidate bindings for e and s arcs, and during the ramification process to test potential bindings of QVISTA nodes as to whether the bindings can satisfy the ElementOf and SubsetOf relationships specified for them in the QVISTA. Hence, these very important classes of deductions are carried out rapidly and "automatically" whenever they are needed, in a manner that requires none of the standard deductive machinery.

6. Derivations Using KVISTA Implications, Disjunctions, and Negated Conjunctions

When the general-purpose binder task has considered all the "explicit" candidate bindings for a given arc, it uses the network's indexing facilities to find nonatomic statements (i.e., implications, disjunctions, and negated conjunctions*) that describe relationships having the same form as the binding being sought. For example, arc E--e-->Builds in Figure 17 is used as the index for finding an implication containing a "build" relationship. Such nonatomic statements are used as the basis for a derivation of the desired binding.

a. Applicability Tests

When such a nonatomic KVISTA statement is found, the general-purpose binder task carries out an applicability test to determine whether the statement can be used to derive a binding for the given QVISTA arc. This test involves unifying (i.e., matching) the KVISTA statement with the QVISTA statement in which the given QVISTA arc is embedded and, when successful, produces a set of substitutions for universally quantified variables that define the "instance" of the KVISTA statement applicable to finding the desired binding.

Several complications in doing the applicability test arise from the fact that neither KVISTA nor QVISTA statements are stored in a canonical form. For example, a negated relationship in the antecedent of an implication can be used to derive a binding for an unnegated form of the relationship, but cannot be used to derive a binding for a negated form of the relationship. In this section, we will discuss the mechanisms in SNIFFER for dealing with these complications.

i. Parity of Embedded Relationships

The applicability tester needs to determine what the logical signs are of the relationships (i.e., terms) that a given KVISTA statement can be used to prove. For example, the statement (~x & y) => (~u v w) can be rewritten in the following ways:

```
(y & u & ~w) => x
(~x & u & ~w) => ~y
(~x & y & ~w) => ~u
(~x & y & u) => w
```

----

*Double negations, negated disjunctions, and negated implications are eliminated from both the KVISTA and QVISTA by simplification rules.

and can therefore be used to prove x, ~y, ~u, or w. If, then, a binding is being sought for a relationship matching x, this statement may be useful in deriving the binding. However, the statement cannot be used to derive a binding for ~x.

The logical signs of the relationships that a given statement can be used to derive correspond to the logical signs that the relationships have when the statement is converted into disjunctive normal form. For example, the disjunctive normal form of the statement given above is x v ~y v ~u v w. The logical signs of x, y, u, and w in this form of the statement are the same as those that the statement can be used to prove.

During the conversion to disjunctive normal form, only two conversion rules change a relationship's logical sign, namely:

~(~x) = x   and   (x => y) = (~x v y) .

Therefore, we can compute a "parity" for each relationship in a statement to indicate the logical sign that it would have in the statement's disjunctive normal form simply by counting the number of negation spaces and antecedent spaces in which it is embedded. The parity associated in this way with relationships allows a quick determination of whether a given KVISTA statement can be used to produce the desired binding.

ii. Parity of Embedded Variables

The applicability tester also needs to determine what type of quantifier (i.e., existential or universal) is associated with each variable in the statement. For example, the statement [~(Ax)P(x) & (Ay)Q(y)] => (Ez)R(z) can also be written:

[(Ay)Q(y) & ~(Ez)R(z)] => (Ax)P(x) and
[~(Ax)P(x) & ~(Ez)R(z)] => (Ey)~Q(y)

and can therefore be used to prove (Ez)R(z) or (Ey)~Q(y) or (Ax)P(x) . If, then, a binding is being sought for an existentially quantified QVISTA node that is a participant in an R relationship, this statement may be useful in deriving the binding. However, the statement cannot be used to derive a binding for a universally quantified node that is a participant in an R relationship.

The quantification types of the variables in the relationships that can be derived from a given statement correspond to the quantification types that the variables have in those relationships when the statement is converted into prenex normal form. For example, the prenex normal form of the statement given above is (Ax)(Ey)(Ez){[~P(x) & Q(y)] => R(z)}. The quantification types of x, y, and z in this form of the statement are the same as those that the statement can be used to derive.

During the conversion to prenex normal form, only two conversion rules change a relationship's logical sign, namely:

```
~(Ax)P(x) = (Ex)~P(x)   or   ~(Ex)P(x) = (Ax)~P(x)
and     [(Ax)P(x) => y] = (EX)[P(x) => y]  or
        [(Ex)P(x) => y] = (Ax)[P(x) => y]  .
```

Therefore, we can compute a "parity" for each variable in a statement to indicate the quantification type that it would have in the statement's prenex normal form simply by counting the number of negation spaces and antecedent spaces in which it is embedded.

Note that this is the same rule that is used for computing the parity of relationships! Therefore, this single, computationally simple rule is used to define a parity for both arcs and nodes. The parity associated with an arc indicates the logical sign of the relationship represented by the arc, and the parity associated with a node indicates whether the node represents a universally or existentially quantified variable.

iii. <u>Matching Embedded Structures</u>

The match process carried out by the applicability tester is a generalization of the ramification process described above and is logically equivalent to unification. An attempt is made to find a set of substitutions that will allow two sets of descriptions to match as follows. The QVISTA contains a description of the relationship that is being sought. When the process begins, a KVISTA statement has been found that describes an existing or derivable relationship. The question being considered is whether a relationship that satisfies the description given in the KVISTA statement will also satisfy the QVISTA description. That question is answered by matching the two descriptions. If the match is successful, it defines a set of substitutions (for universally quantified variables) that must be made in the KVISTA description for it to describe a relationship that would also satisfy the QVISTA description. These substitutions produce an "instance" of the KVISTA statement that can be used as a basis for a derivation. For example, if the selected QVISTA arc is part of the relationship Q(a) and the candidate binding is in the consequent of (Ax)[P(x) => Q(x)], then the instance P(a) => Q(a) would be created.

The basic rules that are used in matching are the following. When comparing the two descriptions, an existential in the KVISTA can match only with an existential in the QVISTA or a universal in the KVISTA, and a universal in the QVISTA can match only with a universal in the KVISTA or an existential in the QVISTA. Remember that nodes that are elements of KVISTA or QVISTA spaces are considered to represent existentially quantified entities. These rules are derived directly from the rules for unification. The key observation is that the derived rules should correspond to the rules used for unification in a refutation proof where the match is being done using the negation of the query.

As an example of the use of parity during an applicability test, consider again the query shown in Figure 16. The general-purpose binder task uses the arc B--e-->Builds as an index to find implication I as a candidate statement to use in the derivation of a binding for the arc Z--e-->Builds. Since both arcs have positive parity, a "builds" relationship derived from the implication will have the desired logical sign. The unification process produces pairings for nodes Z, ?X, and M, and for the obj and agt arcs. All the members of those pairs have positive parity except node M. Node M's negative parity indicates that it is universally quantified and can therefore be paired with an existential KVISTA node having positive parity, namely Ole-Black. The resulting substitution of Ole-Black for M creates the instance of the implication that is used in the derivation.

b. Extracting Embedded Structures

When an applicable nonatomic KVISTA statement has been found, the derivation that is initiated can be thought of as one designed to "extract" the desired embedded relationship from the statement so that the relation or its negation can be asserted at the top level of the KVISTA and the desired binding can be assigned. For example, if the candidate binding is in the disjunct x of a disjunction (x v y), then finding a solution to the subproblem "prove ~y" will allow x to be asserted and the binding to be assigned.

i. Rules for Extraction

The derivation is begun by creating the appropriate instance of the KVISTA statement (as indicated by the applicability test) and then applying the following extraction rules:

To extract	Given	Attempt to prove
x_i	x_1 v ... v x_n	$\sim x_1$ & ... & $\sim x_{i-1}$ & $\sim x_{i+1}$ & ... & $\sim x_n$
$\sim x_i$	$\sim(x_1$ v ... v $x_n)$	x_1 & ... & x_{i-1} & x_{i+1} & ... & x_n
x	y => x	y
~x	x => y	~y

Note that the extraction rules for negated conjunctions and for implications are merely rewrites of the rule for disjunctions.

If an instantiated implication contains a universally quantified variable, then that variable becomes part of the subproblem produced by extracting either the antecedent or the consequent and is free to be bound during the process of solving the subproblem. For example, suppose the original implication is of the form

(Ax)(Ay)[P(x,y) => Q(x,y)] and the instantiation is of the form (Ax)[P(x,a) => Q(x,a)]. If the consequent is to be extracted, then the subproblem has the form "Find an x such that P(x,a)." The assertion that is made when the subproblem is solved is of the form Q(<binding of x>,a).

ii. Nesting

If the relationship being extracted is embedded in a nesting of disjunctions, negated conjunctions, or implications (such as the Q(x) in P(x) => [Q(x) v ~R(x)]), then it is necessary to apply a sequence of extraction rules to complete the extraction. The rules are applied "top down" to the outermost disjunction, negation, or implication first, and all the desired extraction rules are applied before any of the subproblems are worked on. Hence, in the above example, a single subproblem is formed consisting of P(x) & R(x). Solution of this subproblem causes assertion of the desired Q(<binding of x>). Doing the complete extraction in one step results in the extraction rules being applied only once, makes available to the deductive machinery all the constraints imposed by all the subproblems, and allows the subproblems to be worked on in whatever order seems the most advantageous.

iii. KVISTA and QVISTA Extension Spaces

Procedures that carry out derivations such as the extractions described above require facilities for creating subproblems, making assumptions, and asserting derived results. We have used K-NET's partitioning features to create such a set of derivation facilities that are available for use by any binder task. In particular, provisions have been made for adding spaces (called "extension spaces") to the QVISTA or to the KVISTA in an environment. KVISTA extension spaces are used for making assumptions and for asserting derived results. QVISTA extension spaces are used for expressing subproblems.

For example, consider an environment E1 where K1 is the current (i.e., most recently added) KVISTA extension space and a binder task for the QVISTA implication x=>y is initiating a derivation by assuming x and establishing y as a subproblem to be proved. The derivation is initiated by creating an environment E2 that is an offspring of environment E1, adding to the XVISTA in E2 a new extension space K2 containing a copy of x, adding to the QVISTA in E2 a new extension space Q2 containing a copy of y, and attaching a demon to space Q2 in E2. When bindings are assigned to all the elements of y, the demon is fired in the current environment (i.e., the environment in which all of the bindings are assigned) and in that environment the demon removes space K2 from the KVISTA, removes space Q2 from the QVISTA, asserts x=>y in space K1 (the new current KVISTA extension space), and assigns this newly derived result as the binding for the original QVISTA implication.

To maintain the relationship between derived results and the assumptions that were used to derive them, the following three rules are used in creating bindings and asserting results.

The first rule is that in each environment only those binder tasks that are seeking bindings for arcs in the most recently added subproblem are allowed to run. This rule helps prevent duplication of effort among environments and assures that effort within an environment created to pursue a particular derivation strategy will not be spent considering other strategies.

The second rule restricts bindings assigned to elements of any given QVISTA space to be elements of KVISTA spaces that existed at the time the QVISTA space was created. In addition to preventing results derived with the aid of assumptions from being used as if they were independent of the assumptions, this restriction is used to maintain the nesting of quantified variables during derivations, as described in the sections below.

The third rule attempts to assure the widest availability of derived results to as many subproblems in as many alternative proof paths as possible. It specifies that each derived relationship be asserted in the newest KVISTA extension space in the set consisting of the space that contains the statement used to initiate the derivation and those KVISTA spaces containing elements that were used as bindings to solve the subproblem created by the derivation. This rule allows a derived result whose derivation does not make use of the assumptions in recently added KVISTA extension spaces to be added in an earlier extension space and therefore be made available to aid in the solution of subproblems created before the assumptions were made.

iv. Use of Extension Spaces for Doing Extractions

During the multiple-level extraction process, the results of some subproblems may be used in the formation and solution of other subproblems. To make this possible and to prevent a subproblem's results from being used before that subproblem is solved, we maintain the order of the subproblems and their results by putting each one in a separate space and adding those spaces to QVISTA and KVISTA as extensions in the order that the extraction rules are applied. For example, the extraction of $R(y)$ from

$$P(a) \Rightarrow \{(Ax \in X)P(x) \;\&\; (Ey \in Y)[(P(y) \;\&\; Q(y)) \Rightarrow R(y)]\}$$

will cause creation of the subproblem, prove $P(a) \;\&\; P(y) \;\&\; Q(y)$, and will produce the results $(Ax \in X)P(x) \;\&\; y \in Y \;\&\; R(y)$. The results $(Ax \in X)P(x)$ and the existence of an entity y that is an element of Y cannot be used in the proof of $P(a)$, but can be used in the proof of $P(y) \;\&\; Q(y)$. This ordering constraint is maintained by creating extension spaces in the following order:

Q1: a QVISTA extension containing $P(a)$ that accepts bindings from the KVISTA that was current when the extraction was initiated.

K1: a KVISTA extension containing $(Ax \in X)P(x) \;\&\; y\ Y$, the results of proving $P(a)$.

Q2: a QVISTA extension containing $P(y) \& Q(y)$ that accepts bindings from K1 and the initial KVISTA.

Demons are attached to spaces Q1 and Q2 that fire upon completion of the subproblem. Those demons cause spaces Q1 and Q2 to be removed from the QVISTA, space K1 to be removed from the KVISTA, and the cumulative results, $(Ax \in X)P(x) \;\&\; y \in Y \;\&\; R(y)$, to be added to the then current KVISTA extension.

7. Special-Purpose Binder Tasks

The basic SNIFFER includes a collection of functions that form special-purpose binder tasks in addition to the general-purpose binder described above. The most important of these embody the derivation strategies for queries containing disjunctions, implications, and negated conjunctions. In this section we will describe this collection of functions.

a. Proving Disjunctions, Implications, and Negated Conjunctions

QVISTA queries are sometimes nonatomic; for example, consider the questions "Were any Mustangs built by Ford?" and "Are all red Mustangs owned by playboys?". The system's special-purpose binder tasks for nonatomic statements occuring in the QVISTA apply a strategy of decomposing the statement into alternative simpler subproblems using the following rules:

To Prove:	Generate n subproblems of the form:
$x_1 \vee \ldots \vee x_n$	Assume $\sim x_{i+1}$ & ... & $\sim x_n$ and prove x_i.
$\sim(x_{i+1}$ & ... & $x_n)$	Assume x_{i+1} & ... & x_n and prove $\sim x_i$.

To Prove:	Generate the subproblems:
$x \Rightarrow y$	Assume x, prove y. Assume $\sim y$, prove $\sim x$.
$(Ax \in X)[P(x) \Rightarrow Q(x)]$	Create x', assume x' X & $P(x')$, prove $Q(x')$. Create x', assume $\sim Q(x')$, prove $\sim[x' \in X$ & $P(x')]$.

As was the case with the extraction rules discussed earlier, the subproblems created for negated conjunctions and for implications are merely rewrites of those produced for disjunctions. Each binder task selects an order in which to produce its subproblems so that the easier ones are produced first.

Each solution to each of the subproblems produces a set of bindings for the entire original statement being proved. Each time one of these binder tasks is run, it creates a subproblem in a newly created offspring environment and reschedules itself in the parent environment. In the offspring environment it adds a new extension space to KVISTA containing a set of assumptions, adds a new extension space to QVISTA containing an expression to be proved, and attaches a demon to the new QVISTA extension space. When the demon is fired by the solution of the subproblem in the QVISTA extension space, it schedules a task that creates bindings for the entire original expression in the then current environment.

If SNIFFER automatically sought inconsistencies between its knowledge base and assumptions that are made, then it would be sufficient to create a single subproblem from a disjunction, namely, assume the negation of all the disjuncts except one and then attempt to prove the remaining one. However, since SNIFFER does not automatically check assumptions for consistency, we must define two subproblems from a disjunction: one that specifies a disjunct to be proved, say x_1, and an assumption $\sim x_2$ & ... & $\sim x_n$; and a second one that consists only of trying to prove that the assumption made in the first subproblem is false. However, the second subproblem is then attempting to prove the equivalent of the disjunction x_2 v ... v x_n, which itself defines two subproblems, etc. Therefore, in fact, n subproblems are defined and they have the form shown in the rule given above. (Note that in an actual proof it is unlikely that many of these subproblems will be created since what appear to be the easiest ones are established first. Only when the initial ones are found to be difficult to solve do others need to be attempted.)

The subproblem formation rule for implications differs from the rule for disjunctions in that the subproblems created from implications may involve universally quantified variables (represented by nodes that occur in both the implication's antecedent and consequent spaces). In each such subproblem, the nodes representing universally quantified variables are "assumed" in the KVISTA extension space created for the subproblem. They therefore represent an entity in the knowledge vista about which nothing is known except the other assumptions made by the subproblem. If the statement to be proved in the subproblem can be shown to be true about that entity, then it is true for all entities for which those assumptions are true. Such a proof is sufficient to complete the subproblem and therefore prove the implication.

For example, if SNIFFER is attempting to prove that only insecure people own red Mustangs (represented by the implication "if x is a red Mustang, then x is owned by an insecure person"), and the generator for implications creates a subproblem that assumes the implication's antecedent and attempts to prove its consequent, then the assumption for that subproblem would be that some newly created node x' represents an entity that is a red Mustang, and the statement to be proved would be that the entity represented by x' is owned by an insecure person.

b. Function Applications

In a previous section we discussed the procedural augmentation of K-NET through the use of the Applications set. A special-purpose binder task creates elements of the Applications set in the KVISTA by calling the indicated function with the indicated arguments. This binder task is needed when a subproblem is created consisting of the antecedent of a KVISTA implication that describes a "procedural attachment" to the network. Such subproblems describe an element of the Applications set that can be created as soon as bindings are determined for each of the arguments. If the binder task is called before all the argument bindings have been determined, then it attaches demons to the unbound argument nodes that will restart the binder task when all of them have been bound. When all arguments are present, the procedure is called and new network structures are added to the KVISTA to represent the result.

The use of the Application set allows a K-NET to explicitly represent meta-relationships between sets of relationships and the procedures that compute them. If a user has no need to represent such meta-relationships explicitly, then procedural augmentation may be realized much more efficiently through the use of user-supplied binder tasks. For example, rather than include the theorem of Figure 14, a specialist for the SUMS relationship set may be added that knows how to call function PLUS and add new information to the KVISTA as described for Applications.

8. Case Analysis Proofs

There is an important class of problems that the deduction mechanisms we have described thus far cannot solve: those that require a case analysis proof (see Loveland and Stickel [28] and Moore [31]). For example, consider the problem of proving some relation R given a KVISTA containing (PvQ) & (P=>R) & (Q=>R). The mechanisms we have described would go into an infinite loop attempting to solve this problem. What is needed is a case analysis mechanism that will, for example, attempt to prove R in the case where P is true and then attempt to prove R again in the case where Q is true. Since R can be proved in both those cases and the KVISTA contains a statement indicating that either P or Q is true, the problem is solved.

A major difficulty in creating a design for a case analysis proof mechanism is the development of a procedure for defining the cases. Every nonatomic statement in the KVISTA defines a candidate set of cases (e.g., an implication x=>y defines the set of candidate cases {~x, y}). Therefore, the problem of defining the cases can be considered to be one of selecting an appropriate nonatomic KVISTA statement.

We are currently experimenting with the following scheme, which appears to be an effective way of making the selection. It is based on the observation that for a case analysis proof to be necessary, it must not be possible (or be impossibly difficult) to complete a proof without the case assumptions. Therefore, in each case the assumptions must be useful at some point in the proof. The key, then, to defining the cases for a case analysis proof is in the recognition during the attempt to construct a standard proof of the need for each of the assumptions in some potential set of cases.

Disjunctions in the KVISTA, for example, are selected to be the basis for a case analysis proof by recording each time one of the disjuncts is extracted (i.e., contains a relationship that matches some relationship in the QVISTA) during a proof attempt. If all the disjuncts of a particular instance of a disjunction have been extracted, then we can conclude that each of the disjuncts would be a useful case assumption and therefore that the disjunction could be the basis for a potentially successful case analysis proof. The same conclusion can be made when both the antecedent and the consequent of an implication or all the conjuncts of a negated conjunction have been extracted.

When such a "fully extracted" KVISTA statement is found, the first common parent of the environments in which the extractions were initiated is found, and a new task is added to that parent environment's agenda to initiate the case analysis proof. That proof attempts to derive bindings for the portion of the QVISTA for which bindings were being derived each time one of the extractions was done. The task creates an offspring environment and in that environment assumes the first case, establishes the statements to be proved in a new QVISTA extension space, and attaches a demon to the new QVISTA extension. The demon does the same thing for the next case. The last demon asserts the statements that have been proved in each case and assigns the appropriate bindings.

Note that in the example given above, any of the three KVISTA statements could be used as the basis for a case analysis proof (e.g., ~P and R is an acceptable set of cases). Our selection procedure could find any one (or all) of them, depending on the order in which new environments are created in the environment tree.

To achieve completeness, one must also consider cases defined by relationships that occur in the initial QVISTA in both a negated and unnegated form. For example, P(x) and ~P(y) occurring in the QVISTA define a useful set of cases P(<binding of x>), ~P(<binding of y>) when the binding of x along one proof path is the same as the binding of y along another path.

F. CONCLUDING REMARKS

The goal of this research is to provide a unified system that has powerful, general mechanisms and that can be made very efficient for solving the most frequently encountered problems in particular application areas. The central idea is to use specialized representations and deduction schemes where they can be effective, while having a logically complete mechanism to fall back on when the special mechanisms fail.

In producing K-NET and SNIFFER, we have attempted to create convenient hooks for adding specialists, and useful building blocks from which those specialists can be constructed. These hooks include the links to procedures (and hence to other representation structures) that are included in K-NET, the pattern-directed strategy selectors in SNIFFER that are capable of invoking user-supplied tasks, and SNIFFER's agenda control mechanism. The building blocks include the taxonomy derivation functions, the unification machinery, and the facilities for manipulating extension spaces.

We plan to continue our experimentation with various specialist routines, both for the rapid handling of particular types of deduction and retrieval and for the extension of the system to include new types of problem solving activities, including reasoning with uncertainties and about processes. Preliminary experience indicates that the facilities provided make this exploration manageable and productive.

An important goal of future work with SNIFFER is to determine the effectiveness of its control mechanisms, particularly the use of INTERLISP coroutines and multiple-level agendas. The use of these mechanisms to coordinate multiple types of problem solving activities is of particular interest to us, as is the use of heuristics to guide the allocation of resources among the various strategies that SNIFFER coordinates. We have only begun to gain experience in these areas. However, the modular control structure of SNIFFER and the strong cross-indexing of K-NET provide a very supportive environment for future explorations.

VII PUBLICATIONS AND PRESENTATIONS

A. PUBLICATIONS

Fikes, Richard E., and Hendrix, Gary G., "A Network-Based Knowledge Representation and Its Natural Deductive System," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, 22-25 August 1977.

Hendrix, Gary G., "LIFER: A Natural Language Interface Facility," Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, May 1977.

Hendrix, Gary G., "Some General Comments on Semantic Networks," Panel on Knowledge Representation, Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, 22-25 August 1977.

Hendrix, Gary G., "Human Engineering for Applied Natural Language Processing," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, 22-25 August 1977.

Hendrix, Gary G., Sacerdoti, Earl D., Sagalowicz, Daniel, and Slocum, Jonathan, "Developing a Natural Language Interface to Complex Data," to appear in ACM Transactions on Database Systems, ACM, New York.

Morris, Paul, and Sagalowicz, Daniel, "Managing Network Access to a Distributed Database," Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California, May 1977.

Novak, Gordon S. "Representations of Knowledge in a Program for Solving Physics Problems," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, 22-25 August 1977.

Sacerdoti, Earl D., A Structure for Plans and Behavior, Elsevier North-Holland, New York, 1977.

Sacerdoti, Earl D., "Language Access to Distributed Data with Error Recovery," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, 22-25 August 1977.

Sacerdoti, Earl D., "QLISP: A Language for the Interactive Development of Complex Systems," Proc. National Computer Conference, New York, June 1976.

Sagalowicz, Daniel, "IDA: An Intelligent Data Access Program," Proc. Third International Conference on Very Large Data Bases, Tokyo, Japan, October 1977.

B. PRESENTATIONS

10 May 1976	Earl D. Sacerdoti, "Applications of Problem Solving Techniques to Automatic Programming." Automatic Programming Seminar, Computer Science Department, Stanford University, Stanford, California.
8 June 1976	Earl D. Sacerdoti, "QLISP: A Language for the Interactive Development of Complex Systems." National Computer Conference, New York, New York.
21 October 1976	Gary G. Hendrix, "Representing Knowledge in Semantic Nets: A Tutorial." Tutorial Session on the Role of Representation in AI Research, ACM National Conference, Houston, Texas.
13 January 1977	Gary G. Hendrix, "A Tutorial in Natural Language Processing." International Institute of Applied Systems Analysis, Luxenburg, Austria.
13 January 1977	Earl D. Sacerdoti, "Language Access to Distributed Data with Error Recovery." Seminar on Topics in Artificial Intelligence, Computer Science Department, Stanford University, Stanford, California.
20 January 1977	Daniel Sagalowicz, "Applications of Artificial Intelligence to Data Base Management." Seminar on Topics in Artificial Intelligence, Computer Science Department, Stanford University, Stanford, California.
3 February 1977	Gary G. Hendrix, "Current Topics in the Representation and Use of Knowledge." Seminar on Topics in Artificial Intelligence, Computer Science Department, Stanford University, Stanford, California.
7 February 1977	Daniel Sagalowicz, "IDA: An Intelligent Data Access Program." Computer Science Seminar, Northwestern University, Chicago, Illinois.
8 February 1977	Daniel Sagalowicz, "IDA: An Intelligent Data Access Program." Computer Science Seminar, University of Pennsylvania, Philadelphia, Pennsylvania.
8 February 1977	Earl D. Sacerdoti, "Language Access to Distributed Data with Error Recovery," Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts.
11 February 1977	Gary G. Hendrix, "Current Topics in the Representation and Use of Knowledge," and "Applied Natural Language Systems," Computer Science Department, Rutgers University, New Brunswick, New Jersey.

10 March 1977	Daniel Sagalowicz, "IDA: An Intelligent Data Access Program." Seminar, Lawrence Berkeley Laboratories, Berkeley, California.
14 March 1977	Earl D. Sacerdoti, "Language Access to Distributed Data with Error Recovery." ACCAT Program Review, Rosslyn, Virginia.
15 March 1977	Earl D. Sacerdoti, "Language Access to Distributed Data with Error Recovery." Computer Science Seminar, Carnegie-Mellon University, Pittsburgh, Pennsylvania.
14 April 1977	Earl D. Sacerdoti, "Programming Languages for Artificial Intelligence." Computer Science Department, Stanford University, Stanford, California.
19 April 1977	Earl D. Sacerdoti, "Strategies for Automatic Problem Solving." Computer Science Department, Stanford University, Stanford, California.
20 April 1977	Gary G. Hendrix (with Richard E. Fikes), "SNIFFER: A Semantic Net Inference Facility." Xerox Palo Alto Research Center, Palo Alto, California
1-5 May 1977	Gary G. Hendrix, "Current Topics in the Representation and Use of Knowledge," and "Current Topics in Applied Natural Language Systems," Computer Science Department, University of Rochester, Rochester, New York.
25 May 1977	Paul Morris and Daniel Sagalowicz, "Managing Network Access to a Distributed Database." Second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California.
26 May 1977	Gary G. Hendrix, "LIFER: A Natural Language Interface Facility." Second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, California.
15 June 1977	Earl D. Sacerdoti, "Data Base Design for Decision Support Systems." Panelist at National Computer Conference, Dallas, Texas.
23 June 1977	Jane J. Robinson, "Syntax and Language Definition." Nordic Cultural Commission Workshop in Computational Linguistics, Menlo Park, California.
23 June 1977	Gary G. Hendrix, "Semantics and the Representation of Knowledge." Nordic Cultural Commission Workshop in Computational Linguistics, Menlo Park, California.
18 August 1977	Gary G. Hendrix (with Richard E. Fikes), "A Network-Based Knowledge Representation and Its Natural

	Deduction System." Workshop on Automatic Deduction, Massachusetts Institute of Technology, Cambridge, Massachusetts.
22 August 1977	Earl D. Sacerdoti, "Language Access to Distributed Data with Error Recovery." Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts.
23 August 1977	Gary G. Hendrix (with Richard E. Fikes), "A Network-Based Knowledge Representation and Its Natural Deduction System." Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts.
24 August 1977	Gary G. Hendrix, "Knowledge Representation." panelist, Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts.
24 August 1977	Gordon S. Novak, "Representations of Knowledge in a Program for Solving Physics Problems." Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts.
25 August 1977	Gary G. Hendrix, "Human Engineering for Applied Natural Language Processing." Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts.
27 September 1977	Gary G. Hendrix, "Current Topics in Applied Natural Language Processing." Seminar on Artificial Intelligence, Computer Science Department, University of California, Berkeley, California.
29 September 1977	Gary G. Hendrix, "Semantic Network Representations." Seminar on Artificial Intelligence, Computer Science Department, University of California, Berkeley, California.
7 October 1977	Gary G. Hendrix, "Developing a Natural Language Interface to Complex Data." Third International Conference on Very Large Data Bases, Tokyo, Japan.
7 October 1977	Earl D. Sacerdoti, "IDA: An Intelligent Data Access Program." Third International Conference on Very Large Data Bases, Tokyo, Japan.
8 October 1977	Earl D. Sacerdoti, "Research Topics in Interactive Data Base Access." Artificial Intelligence Seminar, Electrotechnical Laboratory, Tokyo, Japan.

REFERENCES

1. "ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report 75-02-08, " FDT (The newsletter of ACM-SIGMOD), Vol. 7, No. 2 (1975).

2. Bledsoe, W. W., R. S. Boyer., and W. H. Henneman, "Computer Proofs of Limit Theorems," Artificial Intelligence, Vol. 3 (1972), pp. 27-60.

3. D. G. Bobrow and B. Wegbreit, "A Model for Control Structures for Artificial Intelligence Programming Languages," Proc. Third International Joint Conference on Artificial Intelligence, Stanford, Ca., pp. 246-253 (August 1973).

4. Bobrow, D. G., and B. Wegbreit, "A Model and Stack Implementation for Multiple Environments," Communications of the ACM, Vol. 16, No. 10. (October 1973).

5. Bobrow, D. G., and T. Winograd, "An Overview of KRL, A Knowledge Representation Language," Cognitive Science, Vol. 1, No. 1 (January 1977).

6. Brown, J. S. and R. R. Burton, "Multiple Representations of Knowledge for Tutorial Reasoning," D. G. Bobrow and A. Collins, eds., Representation and Understanding (Academic Press, New York, 1975), pp. 311-349.

7. Burton, R. R., "Semantic Grammar: An Engineering Technique for Constructing Natural Language Understanding Systems," BBN Report No. 3453, Boston, Mass. (December 1976).

8. Carlson, C. R. and R. S. Kaplan, "A Generalized Access Path Model and Its Application to a Relational Data Base System," Proceedings of the International Conference on Management of Data, Washington, D.C. (1976), pp. 143-154.

9. Chamberlin D. D., J. N. Gray and I. L. Traiger, "Views, Authorization, and Locking in a Relational Data Base System," Proceedings AFIPS National Computer Conference, AFIPS Press, Vol. 44 (1975).

10. CODASYL Data Base Task Group, April 1971 Report (ACM, New York. 1971).

11. Codd, E. F., "A Relational Model of Data for Large Shared Data Approach " Proc. Fourth International Joint Conference on Artificial Intelligence, Tbilisi, USSR (September 1975), pp. 866-872.

12. Codd, E. F., "Seven Steps to Rendezvous with the Casual User," in J. W. Klimbie and K. I. Koffeman, eds., Data Base Management (North-Holland, 1974), pp. 179-200.

13. Computer Corporation of America, "Datacomputer Version 1 User Manual," CCA, Cambridge, Massachusetts (August 1975).

14. Digital Equipment Corporation, "Data Base Management System -- Programmer's Procedures Manual," DEC System Manual DEC-20-AA-4149B-1M (May 1977).

15. Erman, E. D., ed., SIGART Newsletter, No. 61 (ACM, New York, February 1977).

16. Farrell, J., "The Datacomputer - A Network Data Utility," Proc. Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Ca. (May 1976), pp. 352-364.

17. Farrell, J., "RDC: A Program to Run the Datacomputer," Technical Memo, Computer Corporation of America, Cambridge, Mass. (June 1974).

18. Furukawa, K., "A Deductive Question Answering System on Relational Data Bases," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Mass. (August 1977).

19. Grosz, B. J., "The Representation and Use of Focus in Dialog Understanding," Ph.D. dissertation, U. C. Berkeley (May 1977).

20. Hammer, M. and A. Chan, "Index Selection in a Self-Adaptive Data Base Management System," Proceedings of the International Conference on Management of Data, Washington, D.C. (June 1976), pp. 1-8.

21. Harris, L. R., "ROBOT: A High Performance Natural Language Processor for Data Base Query," in [15], pp. 39-40.

22. Harris, L. R., "User Oriented Data Base Query with the ROBOT Natural Language Query System," Proceedings of the Third International Conference on Very Large Data Bases, Tokyo, Japan (October 6-8, 1977).

23. Hendrix, G. G., "Human Engineering for Applied Natural Language Processing," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Mass. (August 1977).

24. Hendrix, G. G. "Expanding the Utility of Semantic Networks through Partitioning," Proc. Fourth International Joint Conference on Artificial Intelligence (September 1975).

25. Hendrix, G. G., E. D. Sacerdoti, D. Sagalowicz and J. Slocum, "Developing a Natural Language Interface to Complex Data," SRI Artificial Intelligence Center Technical Note 152, Menlo Park, Ca. (August 1977) [to appear in ACM Transactions on Database Systems.]

26. Hendrix, G. G., "The LIFER Manual: A Guide to Building Practical Natural Language Interfaces," SRI Artificial Intelligence Center Tech. Note 138, Menlo Park, Ca. (February 1977).

27. Hopcroft, J. E., and J. D. Ullman, Formal Languages and their Relation to Automata (Addison-Wesley, Reading, Mass. 1969).

28. Loveland, D. and M. Stickel, "A Hole in Goal Trees," Proc. Third International Joint Conference on Artificial Intelligence, Stanford, California (August 1973), pp. 153-161.

29. Marill, T. and D. Stern, "The Datacomputer--A Network Data Utility," AFIPS Conference Proceedings, Vol. 44, (May 1975), pp. 389-395.

30. Minsky, M., "A Framework for Representing Knowledge," Artificial Intelligence Memo No. 306, MIT, Cambridge, Mass. (June 1974).

31. Moore, R. C. "Reasoning from Incomplete Knowledge in a Procedural Deduction System", MIT AI-TR-347 (December 1975).

32. Morris, P. and D. Sagalowicz, "Managing Network Access to a Distributed Data Base," Proc. Second Berkeley Workshop on Distributed Data Management and Computer Networks, Berkeley, Ca. (May 1977).

33. Mylopoulos, J., A. Borgida, P. Cohen, N. Roussopoulos, J. Tsotsos, and H. Wong, "TORUS - A Natural Language Understanding System for Data Management," Proc. 4th International Joint Conference on Artificial Intelligence, Tbilisi, USSR (August 1975).

34. Nahouraii, N., L. O. Brooks and A. F. Cardenas, "An Approach to Data Communications Between Generalized Data Base Management Systems." Proc. 2nd International Conference on Very Large Data Bases, Brussels, Belgium (September 1976), pp. 117-142.

35. Paxton, W. H., "A Framework for Speech Understanding," SRI Artificial Intelligence Center Tech. Note 142, Menlo Park, Ca. (June 1977).

36. Reiter, R., "An Approach to Deductive Question-Answering Systems," in [15], pp. 41-43.

37. Roussopoulos, N. and J. Mylopoulos, "Using Semantic Networks for Data Base Management," Proc. 1st International Conference on Very Large Data Bases, Framingham, Mass. (September 1975), pp. 144-172.

38. Sacerdoti, E. D., "Language Access to Distributed Data with Error Recovery," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Mass. (August 1977).

39. Sagalowicz, D., "IDA: An Intelligent Data Access Program," Proc. Third International Conference on Very Large Data Bases, Tokyo, Japan (October 1977).

40. Sowa, J. F., "Conceptual Graphs for a Data Base Interface," IBM Journal of Research and Development, Vol. 20 No. 4 (July 1976), pp. 336-357.

41. Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. International Conference on Management of Data, San Jose, California (May 1975), pp. 65-78.

42. Teitelman, W., "INTERLISP Reference Manual," Xerox PARC, Palo Alto, Ca. (December 1975).

43. Thompson, F. B., and B. H. Thompson, "Practical Natural Language Processing: The REL System as Prototype," M. Rubinoff and M. C. Yovits, eds., Advances in Computers 13 (Academic Press, New York, 1975), pp. 109-168.

44. Walker, D. E., B. J. Grosz, G. G. Hendrix, W. H. Paxton, A. E. Robinson, and J. Slocum, "An Overview of Speech Understanding Research at SRI," Proc. Fifth International Joint Conference on Artificial Intelligence, Cambridge, Mass. (August 1977).

45. Waltz, D., "Natural Language Access to a Large Data Base: an Engineering Approach." Proc. Fourth Internatioal Joint Conference on Artificial Intelligence, Tbilisi, USSR (September 1975), pp. 868-872.

46. Waltz, D., and B. A. Goodman, "Writing a Natural Language Data Base System," Proc. 5th International Joint Conference on Artificial Intelligence, Cambridge, Mass. (August 1977), pp. 144-150.

47. Wilber, B. M., "A QLISP Reference Manual," SRI Artificial Intelligence Center Technical Note No. 118, Menlo Park, Ca. (March 1976).

48. Winograd, T., "Five Lectures on Artificial Intelligence," Stanford Artificial Intelligence Laboratory, Memo No. AIM-246, Stanford, Ca. (September 1974).

49. Woods, W. A., "Transition Network Grammars for Natural Language Analysis," Communications of the ACM, Vol. 13, No. 10 (October 1970), pp. 591-606.

50. Woods, W. A., "An Experimental Parsing System for Transition Network Grammars," in R. Rustin, ed., Natural Language Processing (Algorithmics Press, New York, 1973).

51. Woods, W. A., M. Bates, G. Brown, B. Bruce, C. Cook, J Klovstad, J. Makhoul, B. Nash-Webber, R. Schwartz, J. Wolf, and V. Zue, "Speech Understanding Systems, Final Technical Progress Report," Bolt, Beranek and Newman Technical Report 3438, Cambridge, Mass. (December 1976).

52. Woods, W. A., R. M. Kaplan, and B. Nash-Webber, "The Lunar Sciences Natural Language Information System," BBN Report 2378, Bolt, Beranek and Newman, Cambridge, Mass. (1972).

Appendix 1

EXAMPLE SESSION WITH LADDER

```
@LADDER
Please type in your name: A. C. CAT

Do you want instructions? (type FIRST LETTER of response) No

Do you want to use 2 Data Computers? No

Do you want to specify a current location (default = Norfolk)? No

Do you wish distance/direction calculations to default to GREAT CIRCLE,
or RHUMB LINE? (you can override by specifying in the query) Great Circl

1_What is the current position of the Kennedy?
PARSED!
Parse time: .68 seconds
        * This counts cpu time used by INLAND.
IDA: ((? PTP) (? PTD) (NAM EQ 'KENNEDY% JF'))
        * This is the call to IDA.
Connecting to Datacomputer at CCA1:
        * FAM indicates which computer is being accessed.  The next
        * 13 lines are interactions between FAM and the Datacomputer.
>> ;0031 771108184236   IONETI· CONNECTED TO SRI-KL-22700010
>>  J150 771108184238   FCRUN: V='DC-4/10.00.1' J=3 DT='TUESDAY,
NOVEMBER 8, 1977 13:42:38-EST' S= CCA'
>> !0041 771108184239   DNCTNX: DATACOMPUTER GOING DOWN IN 905 MIN
BECAUSE SYSTEM IS GOING DOWN AT WED NOV 9 77 5:00:00AM-EST FOR 240 MIN
DUE TO SCHEDULED PM
>> ;J200 771108184239   RHRUN: READY FOR REQUEST
*> Set parameters
*<      Exit

CCA1:Z
*> Set parameters
*< V    Verbosity (-1 to 5):   1
*< P    PROCEED with Datalanguage  [confirm with <CR>]
        * The connection has now been established.   FAM now logs in
        * and opens the necessary files.
CCA1:LOGIN %TOP.ACCAT.GUEST ;
CCA1:OPEN %TOP.ACCAT.SAGALOWICZ.NSTDPORT1 WRITE;
CCA1:OPEN %TOP.ACCAT.NTRACKHIST READ;
CCA1:OPEN %TOP.ACCAT.NNSHIP READ;
        * FAM now transmits the query.
CCA1:FOR R1 IN NNSHIP WITH (NAM EQ 'KENNEDY JF') FOR NSTDPORT1 , R2 IN
CCA1:NTRACKHIST WITH R2.UICVCN EQ R1.UICVCN BEGIN STRING1 = R2.PTP
```

```
CCA1:STRING2 = R2.PTD END;
*> Total bytes transferred: 27
IDA = ((PTP '6000N03000W' PTD 7601171200))
        * This is the value returned by IDA.
Computation time for query: 4.077 seconds
        * This counts cpu time used by IDA and FAM.  Extra time is
        * needed to establish the network connection, log in, and
        * open files.
Real time for query: 224.725 seconds
        * This measures real time from the time the request is made
        * to IDA until IDA returns the answer.
(POSITION 6000N03000W DATE 7601171200)
        * Kennedy was last reported to be at 60 degrees North,
        * 30 degrees West, at noon on January 17, 1976.

2_of kitty hawk
Trying Ellipsis:  WHAT IS THE CURRENT POSITION OF KITTY HAWK
Parse time: .97 seconds

IDA: ((? PTP) (? PTD) (NAM EQ 'KITTY% HAWK'))
CCA1:FOR R1 IN NNSHIP WITH (NAM EQ 'KITTY HAWK') FOR NSTDPORT1 , R2 IN
CCA1:NTRACKHIST WITH R2.UICVCN EQ R1.UICVCN BEGIN STRING1 = R2.PTP
CCA1:STRING2 = R2.PTD END;

*> Total bytes transferred: 27
IDA = ((PTP '3700N01700E' PTD 7601171200))
Computation time for query: 1.077 seconds
Real time for query: 78.105 seconds
(POSITION 3700N01700E DATE 7601171200)

3_To what country does each merchant ship in the north atlantic belong
PARSED!
Parse time: .386 seconds

IDA: ((? NAT) (? NAM) ((TYPE EQ 'BULK') OR (TYPE EQ 'TNKR'))
(PTPNS EQ 'N') (PTPEW EQ 'W') ((PTPY GT 600) OR (PTPX LT 3600) OR
(PTPX GT 3900)) (? PTP) (? PTD))
CCA1:OPEN %TOP.ACCAT.SAGALOWICZ.NSTDPORT2 WRITE;
CCA1:OPEN %TOP.ACCAT.NNMOVES READ;
CCA1:FOR R1 IN NNMOVES WITH ((TYPE EQ 'BULK') OR (TYPE EQ 'TNKR')) FOR
CCA1:R2 IN NTRACKHIST WITH (PTPNS EQ 'N') AND (PTPEW EQ 'W') AND
CCA1:((PTPY GT 600) OR (PTPX LT 3600) OR (PTPX GT 3900)) AND R2.UICVCN
CCA1:EQ R1.UICVCN FOR NSTDPORT2 , R3 IN NNSHIP WITH R3.UICVCN EQ
CCA1:R2.UICVCN BEGIN STRING1 = R2.PTP STRING2 = R2.PTD STRING3 = R3.NAT
CCA1:STRING4 = R3.NAM END;

*> Total bytes transferred: 4582
IDA = ((NAT 'US' NAM 'GREENVILLE% VICTORY' PTP '1533N03010W' PTD
7601171200) (NAT 'US' NAM 'TOULE% J' PTP '1533N03010W' PTD 7601171200)
 ...)
Computation time for query: 5.702 seconds
Real time for query: 228.231 seconds
```

```
I have 79 answers; do you want to see them all? ...Yes

NAT SHIP              POSITION    DATE
US  GREENVILLE VICTORY 1533N03010W 7601171200
US  TOULE J            1533N03010W 7601171200
         .
         .      {other entries omitted here}
         .
WG  TRANSQUEBEC        4633N02129W 7601171200
NO  TALABOT            0600N01130W 7601171200

4_the fastest sub
Trying Ellipsis:  TO WHAT COUNTRY DOES THE FASTEST SUB BELONG
Parse time: 1.484 seconds

IDA: ((? NAT) (? NAM) (* MAX MCSF) (TYPE1 EQ 'S') (TYPE2 EQ 'S'))
        * In the interests of brevity and clarity, the remaining
        * interactions with the Datacomputer are omitted.
IDA = ((NAT 'US' NAM 'LOS% ANGELES' MCSF '30.0'))
Computation time for query: 2.317 seconds
Real time for query: 173.3 seconds
(NAT US SHIP LOS ANGELES MXSPD 30.0)

5_what ships are en route to america
Trying Ellipsis:  Ellipsis has failed.
The parser does not expect the word "AMERICA" to follow
"WHAT SHIP ARE EN ROUTE TO"
Options for next word or symbol are:
<COUNTRY>,  <PORT>
end-of-list
        * LIFER cannot parse the question; "America" is not in
        * the vocabulary.

6_what is a <country>
PARSED!
        * In addition to questions about the data base, INLAND can
        * answer questions about the grammar itself.
"<COUNTRY>" may be any sequence of words following one of the patterns:
<COUNTRY> =>  THE <COUNTRY>
<COUNTRY> =>  U S
<COUNTRY> =>  U S S R
<COUNTRY> =>  U S S
<COUNTRY> =>  U S A
<COUNTRY> =>  U K
<COUNTRY> =>  SOVIET UNION
<COUNTRY> =>  UNITED STATES
<COUNTRY> =>  UNITED KINGDOM
<COUNTRY> =>  SOUTH AFRICA
<COUNTRY> =>  WEST GERMANY
<COUNTRY> =>  SAUDI ARABIA
<COUNTRY> =>  GREAT BRITAIN
<COUNTRY> =>  H M S
```

```
"<COUNTRY>" may be any member of the set {ANGOLA ANGOLAN ARABIA ARABIAN
ARGENTINA ARGENTINAN BRITAIN BRITISH CANADA CANADIAN DUTCH EGYPT
EGYPTIAN ITALY LIBERIA LIBERIAN NETHERLANDS NORWAY NORWEGIAN PORTUGAL
PORTUGUESE RUSSIA RUSSIAN SOVIET SPAIN SPANISH U.K. U.S. U.S.A. U.S.S.
U.S.S.R. UK US USA USS USSR VENEZUELA VENEZUELAN}
Finished
7_define america like usa
PARSED!
        * The user may add new synonyms to the vocabulary.
FINISHED
8_redo 5
        * Here we are using the "redo" feature of INTERLISP.
PARSED!
Parse time: .356 seconds

IDA: ((? NAM) (DSC EQ 'US'))

IDA = ((NAM 'KENNEDY% JF') (NAM 'LOS% ANGELES') (NAM 'BATON% ROUGE')
(NAM 'PHILADELPHIA') (NAM 'POGY') (NAM 'ASPRO') (NAM 'SUNFISH') (NAM
'KAWISHIWI'))
Computation time for query: 1.098 seconds
Real time for query: 67.16 seconds
SHIP = KENNEDY JF, LOS ANGELES, BATON ROUGE, PHILADELPHIA, POGY, ASPRO,
SUNFISH, KAWISHIWI

9_how many of them are navy ships
                THEM => ((DSC EQ 'US'))
PARSED!
        * 'Them' or 'she' is currently always interpreted as a
        * reference to a set of ships in the previous query.
Parse time: .505 seconds

IDA: ((? NAM) (DSC EQ 'US') ((TYPE NE 'BULK') AND (TYPE NE 'TNKR')))

IDA = ((NAM 'KENNEDY% JF') (NAM 'LOS% ANGELES') (NAM 'BATON% ROUGE')
(NAM 'PHILADELPHIA') (NAM 'POGY') (NAM 'ASPRO') (NAM 'SUNFISH') (NAM
'KAWISHIWI'))
Computation time for query: 1.205 seconds
Real time for query: 89.417 seconds
8 of them:
SHIP = KENNEDY JF, LOS ANGELES, BATON ROUGE, PHILADELPHIA, POGY, ASPRO,
SUNFISH, KAWISHIWI

10_give status kitty hawk
Trying Ellipsis:  Ellipsis has failed.
The parser does not expect the word "STATUS" to follow
"GIVE"
Options for next word or symbol are:
<RELATIVE.CLAUSE>, <SHIP>, <VALUE.SPEC>, THE
end-of-list
```

```
11_define (give status kitty hawk)
like (list the employment schedule, state of readiness, commanding
officer and position of kitty hawk)
PARSED!
        * This is an example of the paraphrase feature of LIFER.  A
        * new pattern is defined by example.
Parse time: .705 seconds
        * The system answers the query as a side-effect of parsing
        * the paraphrase.
IDA: ((? ETERM) (? EBEG) (? EEND) (? READY) (? RANK) (? CONAM)
 (? PTP) (? PTD) (NAM EQ 'KITTY% HAWK'))

1DA = ((ETERM 'SURVOPS' EBEG 760103 EEND 760205 READY 2 RANK 'CAPT'
CONAM 'SPRUANCE% R  PTP '3700N01700E' PTD 7601171200))
Computation time for query: 2.725 seconds
Real time for query: 173.404 seconds
(EMPLMNT SURVOPS EMPBEG 760103 EMPEND 760205 READY 2 RANK CAPT NAME
SPRUANCE R POSITION 3700N01700E DATE 7601171200)
LIFER.TOP.GRAMMAR  =>  GlVE STATUS <SHIP>
        * The generalized pattern for the paraphrase is added to
        * the grammar.
F0086 (GIVE STATUS <SHIP>)
        * F0086 is the new LISP function created to be the response
        * expression for this pattern.

12_give status us cruisers in the mediteranean
                            spelling-> MEDITERRANEAN
PARSED!
Parse time: 2.855 seconds
1DA: ((? ETERM) (? EBEG) (? EEND) (? READY) (? RANK) (? CONAM)
(? PTP) (? PTD) (? NAM) (NAT EQ 'US') (TYPE1 EQ 'C') (TYPE2 NE 'V')
(TYPE NE 'CGO'))

IDA = ((ETERM 'CARESC' EBEG 760101 EEND 760601 READY 1 RANK 'CAPT'
CONAM 'MORRIS% R' PTP '4000N00600E' PTD 7601171200 NAM 'CALIFORNIA')
(ETERM 'CARESC' EBEG 751231 EEND 760615 READY 1 RANK 'CAPT' CONAM
'HARMS% J' PTP '3700N01700E' PTD 7601171200 NAM 'DANIELS% J') ...)
Computation time for query: 3.738 seconds
Real time for query: 195.698 seconds

EMPLMNT:   CARESC       CARESC       CARESC       CARESC
EMPBEG:    760101       751231       751231       751231
EMPEND:    760601       760615       760615       760615
READY:     1            1            1            1
RANK:      CAPT         CAPT         CAPT         CAPT
NAME:      MORRIS R     HARMS J      EVANS O      FRENZINGER T
POSITION:  4000N00600E  3700N01700E  3700N01700E  3700N01700E
DATE:      7601171200   7601171200   7601171200   7601171200
SHIP:      CALIFORNIA   DANIELS J    WAINWRIGHT   JOUETT
        {information about 8 other ships omitted}
```

```
13_done
PARSED!
File closed  8-Nov-77 11:11:17
Thank you
@
```

Appendix 2

SPECIFICATIONS OF THE INTELLIGENT DATA ACCESS PROGRAM

The Intelligent Data Access program (IDA) is intended to provide a "structure-free" access capability to databases. It uses FAM as its back end in order to also give the impression of using a distributed database management system. IDA is implemented as a set of INTERLISP functions that may either be loaded into the core image of an INTERLISP application program or be accessed via the inter-fork communication capability of TENEX or TOPS-20 by non-INTERLISP applications. It requires the simultaneous loading of FAM in the same INTERLISP package.

Before using IDA, a user must call FAMINIT--see FAM specifications--, and load the "structural schema" which gives a description of the structure of the database. Then, to cleanly finish accesses to the database, a user should call FAMEND. As part of the IDA program, only one command is provided--namely IDA. The description of the IDA command is now given.

IDA(querylst):querylst is a list of lists. Each list may take on one of three possible formats:

a. (? attribute) : this indicates that the user wants to know the values of the attribute for some specified "records" or relation tuples.

b. any boolean expression such as (attribute <OP> value) , where <OP> is

* GT, GE, EQ, NE, LE or LT ; or any combination of those expressions
* using the operators AND and OR. This expression "restricts" the
* records to be examined. A possible example would be: ((LGH LT 1000) AND (LGH GT 300))

c. an expression of the form (* <*OP> attribute) where <*OP> may be MAX, MIN

* or COUNT. The first two cases (MAX and MIN) restrict the records
* of interest to those which maximize the attribute value. The last
* case (COUNT) indicates that one is interested in the number of
* values for the attribute.

The IDA command causes the system to generate a query program in Datalanguage which will execute the query represented by querylist. Once the Datalanguage program is generated, FAMSEND is called with the appropriate arguments. Finally, a list of tuples is returned as the value of the IDA call. Each tuple is composed of a sequence of attribute names and values and corresponds to one "record". The following examples are self-explanatory.

Examples:

```
1_ida(((? nam)(? lgh)(type eq %'cv')))

((NAM 'CONSTELLATION' LGH '1072FT') (NAM 'KENNEDY% JF' LGH '1072FT')
    (NAM 'KITTY% HAWK' LGH '1072FT') (NAM 'AMERICA' LGH '1072FT') (NAM
    'SARATOGA' LGH '1039FT') (NAM 'INDEPENDENCE' LGH '1039FT') (NAM
    'MINSK' LGH '0925FT') (NAM 'KIEV' LGH '0925FT') (NAM 'MOSKVA' LGH
    '0625FT') (NAM 'LENINGRAD' LGH '0625FT'))

2_ida(((? nam)(? lgh)(* max ncsf)))

((NAM 'SVERDLOV' LGH '0689FT' NCSF '18.0'))

3_ida(((* count nam)(type eq %'CV')))

(COUNT 10)
```

Appendix 3

SPECIFICATION OF THE FILE ACCESS MANAGER

The File Access Manager (FAM) is intended to provide a stopgap capability for the access (not creation or update!) of a distributed data base implemented on multiple Datacomputers in an Arpanet environment. FAM is implemented as a set of INTERLISP functions that may either be loaded into the core image of an INTERLISP application program or be accessed via the inter-fork communication capability of TENEX or TOPS-20 by non-INTERLISP applications.

Typical users will only employ the commands FAMINIT, FAMSEND, and FAMEND. For those with special needs, a second level of commands is provided that gives the user more control over FAM, at the expense of some bookkeeping and with many opportunities to bring the distributed file system crashing down.

1. Description of FAM Commands

In the command descriptions to follow, a generic file name refers to a name such as ATLANTIC%SHIP%FILE that refers to a kind of file with a particular kind of information in it, as opposed to a particular file on a particular machine. (In this appendix, references to Datacomputer files apply as well to Datacomputer ports.) Generic Datalanguage is equivalent to ordinary Datalanguage except that references to files (and ports) are replaced by references to generic files (and ports). A ¦location¦ refers to a list whose CAR is the name of a particular machine, and whose CDR is a specific file name. The correspondence between generic files and particular locations is maintained in a model of the distributed file system that is maintained on local disk storage at each facility where FAM is used. The particular file to use as the current model for FAM is determined by an argument to FAMINIT, as described below. References to "the model" in the text below refer to the model used by FAMINIT for a particular session with FAM.

Each command has as a final argument a priority. This priority is a three-digit integer, and currently is ignored. If it is NIL for any command, the default priority specified by the argument to FAMINIT is assumed. Eventually, the priority value will determine the allocation of resources by the remote Datacomputers.

A description of each command to FAM follows:

FAMINIT(¦model¦ ¦user-initialization¦ ¦priority¦) - Initializes FAM.

Causes model of distributed file system to be read from the disk. If ¦model¦ is null, the default model (currently <TRANSACTION>TABLE) is used. ¦user-initialization¦ may be used to specify generic files to be opened at initialization time. If ¦user-initialization¦ is a list, each element is assumed to be the name of a generic file to be opened. If it is atomic, a file by that name is read from disk, and each top-level expression in the list is evaluated. ¦priority¦ specifies the default priority for FAM operations. The default may be overridden in any other command.

FAMOPEN(¦generic-file-name¦ ¦priority¦) - Opens a version of the file specified by ¦generic-file-name¦ for output (exception: ports are opened for input). Returns the location of the opened file.

FAMLOGIN(¦machine¦ ¦login-account¦ ¦priority¦) - Logs in to ¦login-account¦ on ¦machine¦. If ¦login-account¦ is NIL, the default account for the ¦machine¦ (as read from the model) is used.

FAMSEND(¦generic-file-list¦ ¦generic-datalanguage¦ ¦priority¦) - Transmits a version of ¦generic-datalanguage¦ to a machine containing the files on ¦generic-file-list¦. Will log in, open files, and copy files from location to location as necessary. If ¦generic-datalanguage¦ refers to a single file, ¦generic-file-list¦ may be atomic. Returns a list whose elements are the values returned from the datacomputer upon evaluation of the datalanguage.

FAMEND(¦priority¦) - Closes files, deletes temporary ones. Updates the current model on disk.

FAMVERBOSE(¦machine¦ ¦num¦ ¦priority¦) - Sets verbosity on ¦machine¦ to ¦num¦.

FAMVERBOSEALL(¦num¦ ¦priority¦) - For each machine, causes a flag to be set which will reset the verbosity to ¦num¦ when that machine is next accessed.

FAMRESYNCH(¦machine¦) - For the interface to a particular ¦machine¦, flushes local buffers; instructs RDC to wait for more input. Useful when the local interface to RDC gets fouled up.

FAMRESYNCHALL(¦priority¦) - For all machines, sets a flag that will cause the connection to be resynchronized before its next access.

FAMCLOSE(¦generic-file-name¦ ¦priority¦) - Closes the open version (or versions) of ¦generic-file-name¦. Returns a list of locations of the closed files.

FAMDELETE(¦location¦ ¦priority¦) - Deletes a particular file specified by ¦location¦, and updates the model accordingly.

FAMCOPY(¦location1¦ ¦location2¦ ¦priority¦) - Copies file at ¦location1¦ to ¦location2¦. ¦location1¦ is normally a list. If it is atomic, it is assumed to be the name of a generic file. ¦location2¦ is also normally a list. If it is atomic, ¦location2¦ may be a local TENEX file. If it is NIL, the temporary file FAMTEMP is used.

Appendix 4
GLOSSARY

Blue File -- A data base consisting of 14 files of simulated command and control data.

Datacomputer -- A data base management system developed by Computer Corporation of America, designed for interactive access over the Arpanet.

DBMS -- Data Base Management System

FAM -- File Access Manager. Maps generic file names onto specific file names on specific computers at specific sites. Initiates network connections, opens files, and monitors for certain errors.

IDA -- Intelligent Data Access. Presents a structure-free view of a distributed data base.

INLAND -- Informal Natural Language Access to Naval Data. The natural language interface to IDA, which incorporates a special-purpose LIFER grammar.

K-NET -- A knowledge representation scheme based on partitioned semantic networks.

LADDER -- Language Access to Distributed Data with Error Recovery. The complete performance system composed of INLAND, IDA, and FAM.

LIFER -- Language Interface Facility with Ellipsis and Recursion. The general facility for creating and maintaining linguistic interfaces.

MS -- Make Set. The LIFER function for defining a metasymbol as a set of lexical items.

MP -- Make Predicate. The LIFER function for defining a metasymbol as a predicate function.

PD -- Pattern Define. The LIFER function for defining a metasymbol as a pattern expansion.

RDC -- An interface package developed by Computer Corporation of America to run on a local host for interacting with a remote Datacomputer.

SNIFFER -- Semantic Network Inference Facility Featuring External Recourse. A deduction package for reasoning about information encoded in K-NET.

VLDB -- Very Large Data Base